TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

PETR BELYAEV

HIGH AVAILABILITY FRAMEWORK FOR MIX-CLOUD SECURE APPLICATIONS

Master of Science thesis

Examiners: Prof. Jose Luis Martinez Lastra, Dr. Andrei Lobov
Examiners and topic approved by the Faculty Council of the Faculty of Automation and Science Engineering on 6th April 2016

ABSTRACT

PETR BELYAEV: High Availability Framework for Mix-Cloud Secure Applications
Tampere University of Technology
Master of Science thesis, 53 pages, 6 Appendix pages
November 2016
Master's Degree Programme in Automation Technology
Major: Factory Automation and Industrial Informatics
Examiners: Prof. Jose Luis Martinez Lastra, Dr. Andrei Lobov
Keywords: High Availability, clustering, cloud

Having one of the services, such as web applications, databases or telephony systems, unavailable because of a single server failure is very annoying, yet very common issue, especially if the service is deployed on-premises. The simplest way to address it is to introduce redundancy to the system. But in this case the amount of physical machines needed will raise, while their efficiency will drop as most of the services do not use 100% of machine's capabilities. The better way to solve the service availability issue is to logically separate the service from the underlying hardware, balancing the load between instances and migrating them between the physical machines in case of failure. This way is much more effective, but it also contains a number of challenges, such as configuration difficulty and inter-service request routing.

The High Availability (HA) framework discussed in this thesis was designed to mitigate those issues. The key goal solved by the HA framework is raising the scalability and reliability of the service while keeping the configuration as simple as possible. The framework binds together a number of existing technologies, automatically installs and manages them with the single goal in mind: to provide an automated, easy-to-use, reliable, and scalable High Availability solution. In addition, the framework provides a distributed yet unified point of control over the whole installation, regardless of the physical location of components, including cloud and PaaS deployments. The framework is meant to be used by small-to-medium sized enterprises.

PREFACE

This thesis was made in the scope of Multicloud Secure Applications (MUSA) project (`http://musa-project.eu`) at Tampere University of Technology (TUT) FAST Lab (`http://www.tut.fi/fast`) in Tampere in the year 2016. The examiners of this thesis have been the head of FAST Lab, Prof. Jose L. Martinez Lastra, and Dr. Andrei Lobov of Tampere University of Technology.

The system currently serves as an underlying platform for the MUSA project, which is the part of EU Horizon 2020 program. I would like to thank MUSA Tampere team for accepting it into the project, especially Luis Gonzalez Moctezuma and Andrei Lobov, who provided both technical and moral support. I would also like to thank project partners for valuable discussions about cloud technologies and solutions.

Tampere, 23.11.2016

Petr Belyaev, `upcfrost@gmail.com`

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACL Access Control List.

CLI Command Line Interface.

DC Data Center.

DNS Domain Name System.

FOSS Free and Open Source Software.

FQDN Fully Qualified Domain Name.

GUI Graphical User Interface.

HA High Availability.

HQ Headquarters.

JWT JSON Web Token.

RAID Redundant Array of Inexpensive/Independent Disks.

REST Representational State Transfer.

SDN Software-Defined Network.

SLA Service Level Agreement.

VM Virtual Machine.

VPN Virtual Private Network.

VRRP Virtual Router Redundancy Protocol.

1. INTRODUCTION

Every production system always aims to fulfill the Service Level Agreement (SLA), which regulates operational performance characteristics. One of the most vital points of almost every SLA is the maximum permitted downtime of the system. In the modern world, lowering the downtime of the system in case of failure can be critical, and in most cases zero or near-zero downtime is required. From probability theory one can estimate that for general complex system without additional availability enhancement mechanisms, the cumulative availability is a product of availability values of its components. The system's cumulative availability can be enhanced by different means, the brief overview is given in the 2. The system can be called Highly Available if its availability value is higher than the product of availability values of its components.

The use of software-based HA solution is one of the most flexible ways to enhance availability if user does not want to be restricted by the particular hardware. The HA Framework discussed in this thesis also falls to this category, but it has two notable differences from the other solutions. First of all, most of the current HA solutions are meant to be deployed on the local infrastructure. This assumption fails in case of cloud and, especially, multicloud and mixed-cloud deployment, which becomes more and more popular nowadays. This type of deployment enforces additional restrictions on the infrastructure, both logical and physical, which will be discussed further. Second, the implementation of the HA concept is not really standardized. In industry, the company that wants to make its IT services highly available usually just hires someone who knows how to build such system. In addition, many HA implementations involve direct source code modification, or at least HA-awareness on the software side, which makes this concept difficult to integrate for the proprietary software.

1.1 Problem statement

The goal of this thesis was to design an open-source HA system which will operate purely on the application level without any need of the software source code modification. In addition, the system should be easy to use, and it should be designed to run smoothly in cloud environments. By combining these points, the resulting system should provide all HA features to cloud-deployed application, while saving time, effort, and money by making system adoption as simple and fast as possible.

1.2 Cloud deployment and clustering

What is the 'cloud'? Multiple definitions can be found, but it should be first stated that this thesis is dedicated to 'cloud hosting', not 'cloud computing'. The wiki gives one more synonym to the word 'cloud', the "on-line computing". The main idea of this concept[8] is to outsource the resources from the user's point of view, and to provide shared, on-demand resources from the provider's point of view. User wants to be able to access these resources from any physical location, and for him the resource should be seen as a single point of interaction. Provider is trying to meet the user expectations, building the decentralized infrastructure with the unified yet distributed access point. Even though this thesis work stands on the provider's viewpoint, the user's point of view should be always kept in mind, since the customer is the main life-source for the business.

Strictly speaking, from the provider's point of view, 'cloud' is a concept which means the abstraction of the resource's logical location in relation to its physical location, and, in many cases, the abstraction of the resource's dedicated machine time in relation to the total machine time available to the provider. The 'cloud' service, for example the Virtual Machine (VM), should be always accessible by the same logical address, but in the real world it can migrate between different servers, rooms, data centers, and continents. This migration should be performed in the seamless way, meaning that the user should not have a clue that the service just migrated somewhere. Such abstraction can be built on the different levels.

On the network level, it can be represented as a double-layered Software-Defined Network (SDN)[7]. This approach should be always kept in mind, since if the hardware is the flesh, then the network is the blood of any IT system. The main idea of SDN is to overcome limitations of standard routing and host naming tools, such

as DNS (Domain Name System), by utilizing high performance key-value storage replicated across the whole infrastructure.

On the application level, it is usually represented as a set of the redundant, state-aware nodes working together. This definition is very close to the definition of the cluster, which is usually defined as a set of the interconnected computers working together in a way that they can be seen as a single computer from the user's viewpoint[21]. In fact, if consider cloud as a data center, then the cloud service will become the main part of the Data Center (DC), the cluster. There is one more viewpoint on this matter, that is stated in [4], which is written by Mosix system developers. From the authors point of view, the cloud is a collection of private clusters, and not a single cluster. In a way, this claim is arguable, as cluster can be formed over the internet, especially in multi-cloud environment. This system is quite interesting, and it will be used as one of the references for the HA framework concept. In the Mosix' administrator guide[3] the authors state that the system provides load-balancing and process migration, it is capable of monitoring and resource management, but it does not provide high availability. In general, the Mosix system is a very good example of the evolution of clustering. It started as a Unix kernel patch-set in 1977, and in over 40 years it evolved into the resource manager which do not require kernel patching, similar to Pacemaker and Torque. The history and classification of clustering systems will be covered in chapter 2.

How to build the cluster, or, to be precise, the clustered infrastructure. Clustering is always a custom case. Many companies provide their own clustering solutions, addressing only some parts of the existing cases, e.g. the clustering of network devices or the database clustering. It means that there are not that many general solutions. This number decreases even more when scaling out of a single room. Mixture of the private cluster in the local server room, and the public cloud represented as a rented VM, can become a real challenge. The way how this challenge is solved by both research and industry will be discussed further in chapter 2. To make the explanation a bit clearer, the following case example will be given to illustrate how the company can face the need of using clouds and clustering.

1.3 Case example

Imagine the company N, which runs its business in the field of medicine. Medical companies are particularly interesting since they have some special data storage

requirements, e.g. they cannot store customer's personal data in the non-trusted environment. The 'trusted environment' is usually the company's local infrastructure.

The company started a decade ago as a single small clinic, with the total personnel counting 100. The IT infrastructure used by the company was adequate compared to the given time and scale – local mail server, local telephony system, local website server, etc. After half a decade, the company has grown up, it now consisted of the Headquarters (HQ) and three branch offices (clinics) with the personnel of 400 workers. The IT infrastructure, originally built for the since office, was scaled using a number of thin clients and VMs, forming a star-shaped pattern with an HQ as a single center of the star.

Suddenly, the company won the government competition for building a regional pharmaceutical factory. Due to the excess amount of money given to the company, it expanded quickly, consuming some smaller companies and giving out a franchise contracts to the others. After just a few years, it became a medium-sized enterprise, with 3000 personnel, two HQ offices, 11 branches, two storehouses, and a factory. The IT infrastructure, originally built for the since office, tried to scale out the current extend and faced to violent problems, namely the lack of resources, the inability to scale, and the lack of stability.

For example, the lack of internet connection in one of the HQs could completely paralyze the whole business process, since the workstation terminal server will become unreachable. On the network level it can be solved by contracting the second ISP, but if the server itself will go down, the network redundancy will not help. Most of the applications used by the company were designed in a way that they did not support native clustering, so in case of a server failure, all the clients should manually redirect their terminals to the backup server. It does not take much time to reconfigure a single application, but it still interrupts the normal business process, and of course it annoys users.

The on-site redundancy can give some stability boost, but it would be still limited by the site's own reliability. And it costs a lot. When the company faced the need in yet another expensive high-end chassis for the second HQ, they decided to migrate some of the services to the local cloud provider, which gives quite a high SLA and reasonable prices. Of course, the company did not want to migrate all of the services into the cloud. First of all, they could not, since the provider was not certified for

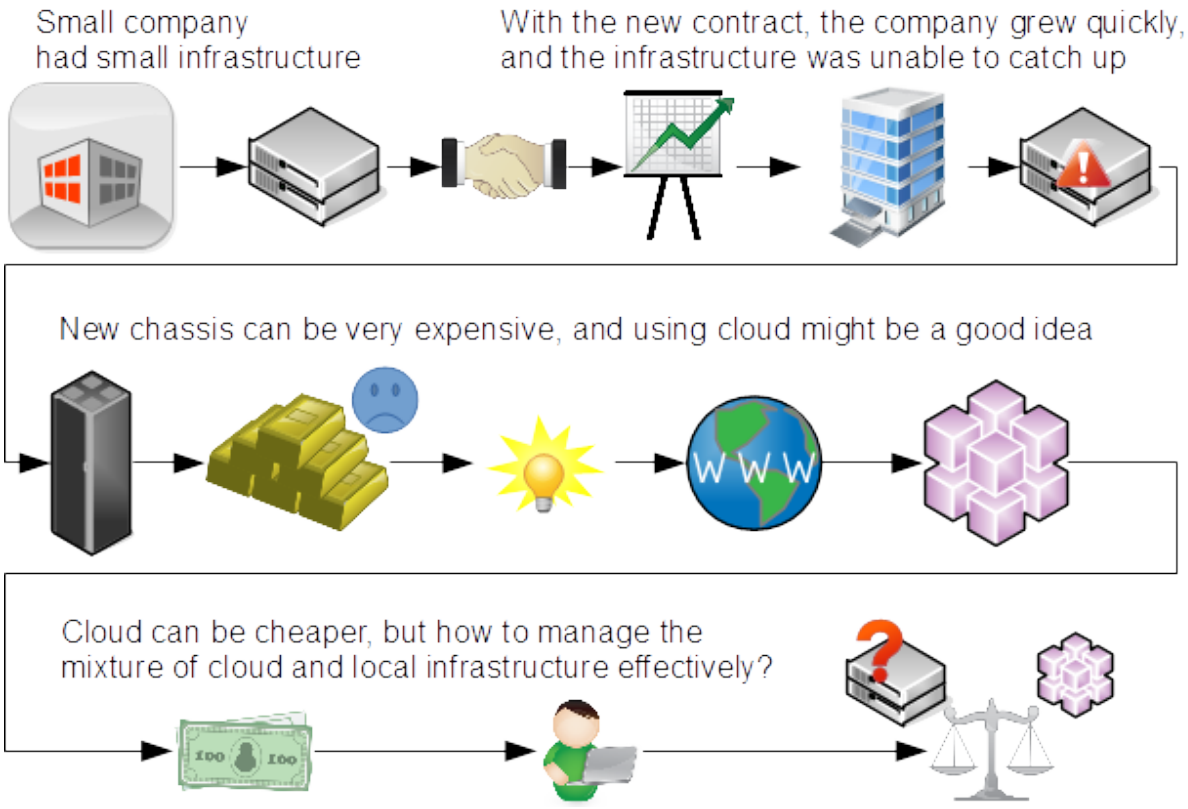

Figure 1.1 *Schematic representation of the case*

the medical personal data operation. Second, the company did not want to throw out the infrastructure that was already deployed, since a few hundreds of thousands of Euro were invested into this infrastructure. Third, migrating some of the services would be just unreasonable. For example, internal telephony system serving 3000 clients every day is a huge traffic hog, and trying to deploy it into the cloud would become a disaster.

The company contracted the local cloud provider for a few VMs of a standard cost rate, intending to deploy some new services on those machines. Those VMs, being a standard issue, have a 'burst' option, which dramatically increases the VM's capacity when switched on, but this feature costs a lot, and the company does not want to use it for some minor accidents. Still, in case of the perfect storm, the company intends to migrate some critical services (e.g. the call center or the production line control) into the cloud, utilizing this feature.

Drawing the line to the given case, on the paper it looks nice and easy. But the main point is that the company's IT department would need to implement a vast

automation layer, to address both fail-over and control cases. The control means that the resulting private-public infrastructure should be controlled as a whole, otherwise it would not allow the seamless automated migration between the server room and the cloud. The fail-over case means that the client should never think to which address to connect, or where the server he is trying to contact is located. And in case of a failure, the client should be able to automatically reconnect to the new server without any additional actions.

2. BACKGROUND

Availability and fault-tolerance issues can be critical not only in the IT field. In mechanics, the simplest example of fault-mitigation mechanism is the reserve parachute, that should be used if the main one failed to open. This principle of having the secondary system is called "redundancy". In electronics and IT, redundancy is usually achieved by introducing additional devices and/or additional software into the system. It addition to redundancy, the system should be able to perform self-diagnostics to migrate to the additional circuit/software instance in case of failure. Thus, most of them can be divided into two categories: hardware-based and software-based. Both types share some of the methods they use. The main method to enhance the availability of the system is to introduce some degree of redundancy. If the element of the system is out of service, it is out, and it will take some time to fix it or change it. To allow system to stay operational regardless of the component failure, the most common, and, in most cases, the only, way is to add the second instance of the same component, which will come up when the first instance will fail. This mode, just as in case of the parachute example, is called "active-passive", and it was popular when HA systems started to emerge. Since this method gives unequal load for two equal components, and since the second component still needs some time to start, nowadays the more popular way is to use "active-active" mode, with two components running at the same time, and to use load-balancer to equalize the load between them.

Historically, hardware-based solutions were the first ones to be developed, as they existed in time when electricity was yet to be discovered. Nowadays, hardware-based solutions are still very popular in networking and mechatronics, mainly because of their reliability and high performance. They usually come in the form of pluggable cards, cables, chips, or devices. In mechatronics, a good example of the redundant hardware-based availability solution is the additional electric motor wiring described in [6]. Two separate coils with two separate voltage supplies allow the motor to continue operation even in case if one of those will break down. For IT and networking,

one common availability example is the HDD Redundant Array of Inexpensive/Independent Disks (RAID)[15] with the hardware RAID controller (figure 2.1). For example, for RAID levels 1, 6 and 10, for each written block, the controller issues writes to the number of connected disks according to the required redundancy level. When the read request arrives, the controller starts to read from multiple disks at the same time, achieving higher read speeds. If one of the disks fails, the controller marks this disk as failed and continues to use the remaining disks. When the disk comes back after being fixed or replaced, it is re-synchronized with the rest of the RAID.

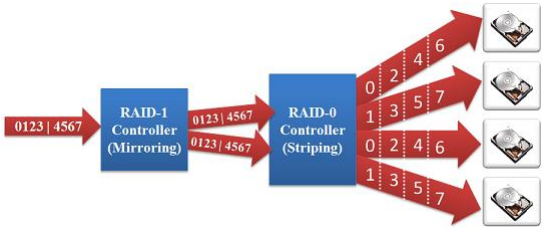

Figure 2.1 *RAID10 operation[10]*

Software-based solutions development began as soon as first PLCs were developed. Two main approaches formed with the evolution of those system: state recovery[2] and redundancy[19]. First approach was typical for real-time systems and PLCs, especially in cases where PLCs were implemented to substitute mechanics, for example in aviation[13]. The second approach, the redundancy, was more popular in cases where the integrity and safety of data were needed the most, namely in storage systems. Modern systems usually implement both of those mechanisms, using redundancy for continuous operation and state recovery for faster failure resolution. One of the most typical examples of the state recovery is the container-based system which redeploys the container in case of failure.

Software-based systems can be subdivided in many different ways. One way is the application-wise subdivision described in [16]. This classification still holds well, even though the recent advances in container-based virtualization shifted the scale quite a bit. The only place which can be somehow corrected is the special place of storage replication systems. The DRBD, mentioned in the paper, in most cases gave up his low-level positions to hardware-based implementations, while higher-level replication systems, such as GlusterFS and GFS2, are closer to the application level. Thus, software-based solutions can be subdivided into OS-level and application-level ones.

OS-level solutions evolved greatly over time. They started with Single System Images (SSI), when the machine was joining the cluster during the boot phase, and the cluster itself was operated as a single meta-machine. At some point, the word "SSI" even became a synonym for the idea of creating a visibly single service with multiple real backends. One quite old, but still good overview of that period can be found in [17]. Good examples of an SSI solution are the openMosix Linux kernel patchset and the Gobelins system[14]. This type of solutions provides so-called "hard load balancing", or the "real load balancing", since the OS kernel can see the exact system load from all running processes. The biggest obstacle for those systems is a very high complexity, as well as some stability issues. When consumer-grade CPUs with hardware virtualization functions came to the market, most of the SSI systems were already collapsing under their own complexity. Before hardware-backed virtualization, virtual machines and hypervisors had a great performance overhead, with only a few mainframe-grade exceptions such as IBM mainframes. Still, some techniques, such as virtual containers with the common OS kernel, were already showing their benefits. This can be seen from [12], which provides the comparison between SSI-based openMosix and hybrid Kerrighed, which utilized some of the container concepts.

Modern OS-level solutions can be divided into hypervisors, virtual machines, and Linux container orchestrators, such as OpenStack or Kubernetes. In most cases, services are wrapped into a single image file which contains the minimal OS distribution inside, which is in a way similar to SSI. In case of VMs, it might also contain the hardware specification required to run the service. The main difference is that the image itself does not become a part of a cluster. Instead, the orchestrator is distributing those images across the cluster. This method simplifies the migration and the quota definition for each service, but it also introduces a lot of overhead, especially for network communication, as host system will behave as a virtual router. Virtualized systems cannot share resources to the same extent as SSI ones, but in many cases their behavior is more stable, they are simpler and in many cases operate faster. Unfortunately, not so many comparison test results between SSI systems and modern virtualized solutions can be found, mainly because most of the SSI systems' development stopped even before the hardware virtualization was introduced. SSI systems are still used for real-time applications, though their number is very limited. From the SSI systems list in Wikipedia[1], a single general purpose SSI system that is still developed actively developed can be spotted, the Mosix system. But according

[1] https://en.wikipedia.org/wiki/Single_system_image

to the changelog[2], starting from version 4.0, Mosix system no longer requires any specific kernel patches, meaning that the development team also dropped the general purpose kernel-level clustering idea and migrated to the application level. One of the challenges which strikes the SSI systems much harder than their orchestrator-based counterparts can be extracted from the Torque resource management system administrator guide[1]. Cluster is a Non-Uniform Memory Access (NUMA) system, which means that the memory access time within the cluster can vary greatly. This feature is not a problem for orchestrator, which originally sees a cluster as a collection of physical machines, but it can pose a problem for a single meta-machine approach, as some parts of memory of this meta-machine can be much further away, in both physical and logical sense, from the target CPU than the others.

Application-level solutions can be both generic and application-specific. The HA Framework discussed in this thesis is a generic solution, build on top of the Corosync[3] and Pacemaker[4] suite. Cluster resource managers, such as Pacemaker and Torque, can serve as good examples of application-level generic solutions. Also, most of the load-balancing proxies, such us LVS (Linux Virtual Server) subsystem, can be also addressed to this category. Generic solutions usually provide process status monitoring and issue start-stop-restart commands to the controlled processes. Since this type of systems has only limited information about the process itself, it usually requires additional input about the nature of the process to operate correctly. In a way, container orchestrator can be seen as a bridge between generic application level and OS-level solution, as it can be seen as an OS-level from the guest's point of view, and as an application for the host.

Application-level application-specific solutions are, in most cases, embedded into the software. These solutions are purely application specific, but they usually do not require any additional configuration from the user. They can also monitor the internal consistency and performance of the process, and react accordingly. Good examples of the application-level solutions are the database clustering systems, such as PostgreSQL clustering mechanism. Those systems not only interconnect the database instances across the cluster and monitor their health, they also perform as a software RAID system, balancing the load and checking the consistency of the database.

[2]http://www.mosix.cs.huji.ac.il/txt_changelog.html
[3]http://corosync.github.io/corosync/
[4]http://clusterlabs.org/

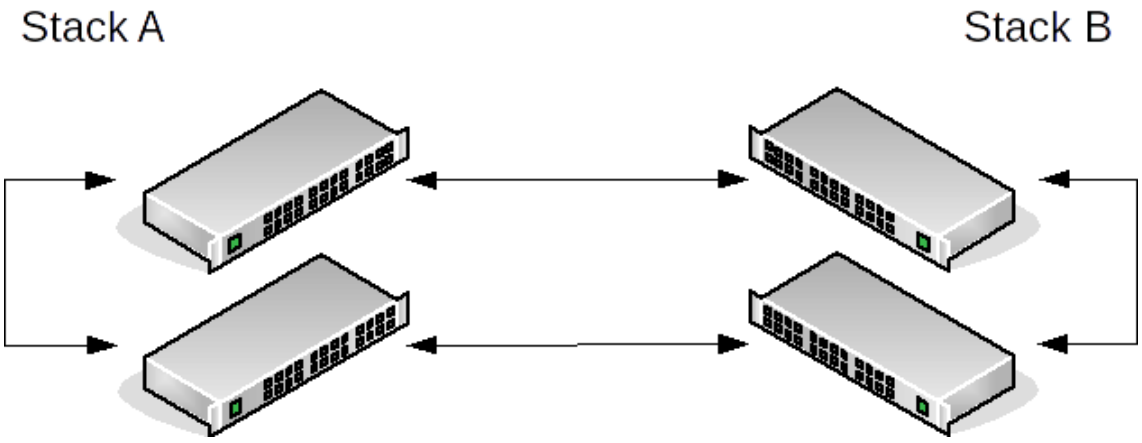

Figure 2.2 *Cross-stack Link Aggregation (EtherChannel)*

Some systems introduce mixed hardware-software-based behavior. One of the examples is the operation of Cisco switches with the Stack Module and link aggregation (EtherChannel) enabled (figure 2.2). In the simplest case, two pairs of switches, each pair interconnected with the stack cable, are connected to each other with two cables. During the normal operation, the load is balanced between the cables using the enhanced Ling Aggregation Control Protocol (LACP) called EtherChannel. If one of the switches fails, its port goes down, and the system forces the inter-pair communication to use only the remaining cable, giving time for the network engineer to fix the faulty switch without interrupting the system operation.

From this retrospective it can be seen that most of the modern solutions can be divided into the following categories: specialized ones (hardware- or application-specific), general purpose resource managers, and VM/container orchestrators. First type has the best performance, but it is limited in application. Orchestrators provide good availability, they are rather simple and easy to control, but they usually introduce a lot of overhead. It is also not always possible to create a container for the application, especially if it requires direct access to the hardware, as in case of telephony systems accessing line cards. Resource managers provide good balance between performance and availability, but in most cases they are harder to configure when compared to orchestrators and specialized solutions. There is a couple of general-purpose systems on the market, and most popular ones are Torque, Mosix,

and Pacemaker with Heartbeat or Corosync used as a clustering engine. The main problem of those systems is that they provide only some of the building bricks, but not the complete solution, and they can be quite hard to comprehend in the zeitnot mode.

In addition to the configuration and management difficulty, there is one more issue that is common not only for resource managers, but for most of the clustering solutions. In the modern world, network routing might become a problem, especially in multicloud setup. Most of those systems, such as Mosix from the resource management side[3] or Docker Swarm with Flannel extension from the orchestrator side, rely on socket forwarding and virtual IPs to migrate the network service. This approach introduces very small overhead, but its application is limited if some parts of network lie outside of the managed infrastructure. This matter will be further discussed in chapter 3.

Currently, main consumers of availability, cloud, and clustering technologies are medium- and large-scale enterprises of all kinds, both related and not related to IT. The main factor pushing the company to the HA practices implementation is usually the size, not the particular industry. Most popular ways to increase availability of the service for those companies are outsourcing to the nearest datacenter, and keeping multiple service instances with both automatic and manual switching for in-house installations. Outsourcing means delegating the availability and clustering problems to the cloud provider, which is not always an option. Manual switching, as well as frequently used automatic switching methods such as DNS-based switching[5, 9] and Virtual Router Redundancy Protocol (VRRP)[11, 22], has a number of drawbacks (see section 3.1), including manual reconnects, DNS caching and constantly annoyed user. Commonly outsources services are websites and, sometimes, non-critical databases. Critical or traffic-hungry software, such as communication and accounting systems, are rarely outsourced for bigger companies. For smaller ones, counting 5-15 workers, almost the whole infrastructure can be located in the cloud. For cloud providers, the most popular solutions are container-based ones, as provider's main concerns are service isolation and resource consumption limitation. For these tasks, containers and virtual machines are the best instruments currently available on the market, and it is quite hard to find a better solution. Thus, the HA framework aims on those companies that prefer to keep their infrastructure in-house.

Combining the information from administration manuals, descriptions and articles related to all software systems mentioned so far, a collection of statements on how the system should be build and how it should operate can be made.

- All the nodes should be state-aware of each other.
- Single yet decentralized logical resource (application) control center should exist.
- The system should know where the requested service is located.
- The service itself can be oblivious to the presence of the cluster.
- The service should not be modified in any way to operate within the cluster.
- The system should provide load-balancing between the service nodes.
- The system should be capable of ad-hoc operation.
- The system should be secure.
- The system should be easy to deploy and configure.

Some of those claims can be solved using the existing software to some extent. Orchestrators allow the service to be oblivious of the cluster existence, and most of the resource management systems provide decentralized control over the setup. Yet, one trying to resolve all those claims will face some limitations from most of the currently available systems. To overcome those limitations is the main goal of the HA Framework discussed in this thesis. It aims to provide a simple, complete, and feature-rich solution to build the highly available server infrastructure in the multi-cloud environment. It is built on top of the Corosync+Pacemaker pair coupled with additional software to provide easier deployment, configuration, and interconnection of the applications. As it goes by the name "framework", it does not force user to use all of its components, allowing to connect standalone non-HA applications, which use only part of the system, with the ones completely managed by the framework.

3. DESIGN OF THE SYSTEM

Design of any complex system takes a number of steps. This section contains step-by-step explanation of how the system was designed and which aspects were taken into account.

3.1 Planning

According to the list made before, the system should be made of state-aware nodes with resource location-aware management system. It also should be non-invasive for the applications deployed on top of it, and it should provide at least some basic security. Let us start with more verbose definition of functional parts of such system.

3.1.1 State-awareness

State-awareness means the communication between the nodes. The very basic check of the state of the system is a live-dead check. Of course, a simple ping can do the trick, but usually more meaningful messages are required. Such possibility on the general application level is provided by the 'heartbeat' application, which is currently superseded by Corosync. Both HB and Corosync are the typical cluster messaging buses, they can check the status of the node and pass the message from one node to another.

3.1.2 Resource management

The message bus can pass the message, but it is up to the resource manager to make this message meaningful. There are not so many choices here. If not using the Mosix stack, two most popular systems will be Torque and Pacemaker. Both systems are general purpose resource managers, which can start, stop and monitor

services across the cluster. Torque is more job-oriented, while Pacemaker is mainly focused on services. As HA framework aims to be used with web services, the Pacemaker will be a better choice. To manage services and resources, this system uses so-called OCF scripts, which resemble the Linux daemon start-stop scripts with some additional features, e.g. the monitoring of the web server can be done not only by the status command, but also by requesting the index page of the website and comparing it to the page that the service administrator anticipates to get.

3.1.3 Resource-location awareness

Resource-location awareness means that the system knows where the resource is located. It can sound simple, but the actual location of the resource inside a vast, highly dynamic environment can be quite hard to predict. In many cases one can retaliate to use of the virtual IPs. It is quite a nice and simple way, but it has one major drawback. It works well only when operating inside a single, secluded and properly routed network. It would be quite hard to make it work in the mixed private-public cloud environment. In many cases, Virtual Private Network (VPN) can be used to establish a single subnet across the cluster, but VPN can be blocked, can be not always legal, and, of course, it adds overhead. The other problem with VPN is the requirement to have access to the infrastructure, which can to always be fulfilled, especially when some services are deployed on PaaS. In fact, it can solve some of the problems like encryption, but it will introduce many additional questions, so it is better to put this solution aside while still keeping it in mind.

DNScan also be used, but that is not the best idea for the real-time service, as DNS takes a lot of time to renew the global directory (up to 24 hours). In many cases it is also cached on the user devices, meaning that even if the global A-record is updated, the user will still try to resolve the Fully Qualified Domain Name (FQDN) from the cache. Thus, DNS is not the best solution for the dynamic system.

Previously, a comparison with the double-layered SDN was made. This comparison can help to draw the logic of the system to find better solution. Basically, some kind of runtime key-value storage which maps service to its location or locations is required. Databases can be used for this task, but it is much easier to use simple and fast Directory servers. There is a number of such applications, and the most stable and well-known of those is the Apache project's Zookeeper. But since it is, first of all, written in Java, and, second, it usually requires the direct modification

of the source, it would violate the non-invasive approach of the final solution. Also, it provides a huge amount of features that are not really needed in this case, such as semaphoring or queues. Having additional features usually means additional resource consumption and, as a consequence, slower operation on lower-powered virtual machine.

Out of lightweight directory services, Etcd[1], Consul[2] and Doozer[3] are the first ones to think of. The Doozer is quite dead, so the most feasible solutions at the moment are Etcd and Consul. Both of them are very lightweight directory services, and they both provide a nice Representational State Transfer (REST) API.

3.1.4 Non-invasive operation

In many cases, the application is not designed to be aware of cluster's existence. It means that an additional layer of abstraction should be made to fence the application from the cluster infrastructure. Let us try to draw the points of interaction between the application and the cluster components. Start-stop is managed by the resource manager, and the application does not really need to know who exactly is pressing the trigger, so it is not a problem. But the connectivity with the others can become one. Even if the system knows where the requested service is, this knowledge should be passed to the application. Aside from the virtual IP method, it would mean that the connectivity abstraction is needed. This abstraction should take in the static request from the application, perform the directory lookup, and redirect the request to its destination. And it would be nice if this redirect will be carried out in a load-balanced way. Thus, load-balancing reverse proxy is a good solution in this situation. Two main solutions would probably be HAproxy[4] and Vulcand[5]. The first one is more stable, but it relies on the static config, which should be regenerated when the topology is changed. Actually it touches the other point of the list, namely the configuration, so this question will be covered a bit later. Vulcand is the second possible solution, it can take the data directly from the directory, so it supports the runtime reconfiguration.

But of course someone should put the data into the directory first. And this is yet

[1] http://clusterlabs.org/
[2] https://www.consul.io/
[3] https://github.com/ha/doozer
[4] http://www.haproxy.org/
[5] https://github.com/vulcand/vulcand

another configuration problem, which will be discussed later. So, non-invasive methods have already been covered, as well as the load-balancing feature of the reverse proxy, so the ad-hoc operation is left to be discussed. This is quite a hard topic, since the ad-hoc operation of the cluster is not really automated in any framework, and it will take a lot of scripting. The basic idea is to provide the new node the address of at least one of the alive cluster nodes, authenticate against this node, send all the cluster nodes the new node address and give the new node all the addresses of the existing nodes.

3.1.5 Runtime configuration

The configuration should be managed in runtime, and some services should be restarted when the configuration changes. That can be quite a pain. To manage the configuration, the application called Confd[6], from the CoreOS[7] project, can be used. It integrates with the directory, takes config templates and populates them with the directory-driven values. It can also restart the services when needed, but this restart should be of course tied down to the resource manager to avoid a split-brain situation. The other possibility is to use directory-specific configurators like Consul-template[8]. Some of the deployment systems can also be used for this task, for example Chef[9] can handle the configuration file generation. This possibility should be taken into account.

3.1.6 Deployment

Deployment can be carried out in many different ways. Chef or Ansible[10] can be used to provision the node from the central server, or one can just write a number of shell scripts to run on each node. Each way has its pros and cons, but in this thesis Chef will be used for deployment. It is a big, stable and popular system. And agile, which is very important. It can be also used as a configuration files generator, but it will take a number of adjustments in the cookbooks to tie them down to the directory, which is not the best idea since the amount of cookbooks is

[6] http://www.confd.io/
[7] https://coreos.com/
[8] https://github.com/hashicorp/consul-template
[9] https://www.chef.io/
[10] https://www.ansible.com/

quite high. The main problem would be not fixing those cookbooks, but maintaining them synced with upstream.

3.1.7 Security

IT security as a topic is too wide to discuss in terms of a master thesis. For now, just a few statements based on the "fundamental rules of the IT security"[18, 20] will be made.

- Rule 1: If someone else has a physical access to your personal computer, this computer is not personal anymore. Unfortunately, cloud will never be completely secure. Even if the system is flawless, one can always reason out with the cloud provider's hardware engineer, for example with bribes or violence. Or, probably, by using both. For user it means that there should be a strict and straightforward control over what kind of data he or she can put into the cloud and what data should always remain on the user's local infrastructure, even if it will go down.
- Rule 2: It is better to use well-known security protocol instead of creating a new one, unless there is a well-justified reason to do so. Solid, major security protocol should not only be implemented in a very careful way, but it also should be tested by hundreds of experts and thousands of users over several years. As the development of the new security protocol is not considered as one of the outcomes of this project, the HA Framework will only use well-established and well-known protocols.
- Rule 3: Service should always know whom it is talking to and were the data comes from. This problem is the authentication problem, and one of the most common solutions to it is to use the X.509 secure certificates.

With all this being said, it is time to start putting everything together by drawing the interaction schemas and process diagrams.

3.2 Basic design

Let us start by simply putting all the components mentioned above on the empty form. So, in the beginning was the Service. Service was published on some node in

a stand-alone way (figure 3.1).

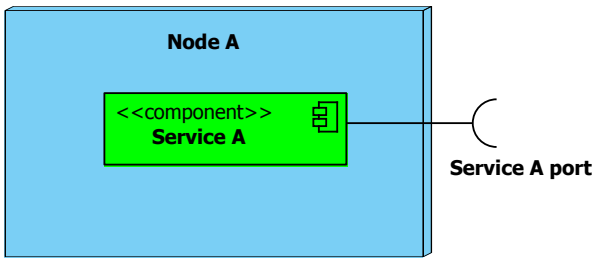

Figure 3.1 *Initial stand-alone service setup*

The cluster should be aware of the service state, which adds cluster messaging and the resource management to the model. This very basic model of a cluster is frequently used in the local deployments (figure 3.2). It can already handle monitoring and launching services, and, as virtual IP can be also seen as a service, this simple system can handle VRRP-based active-passive cluster operation.

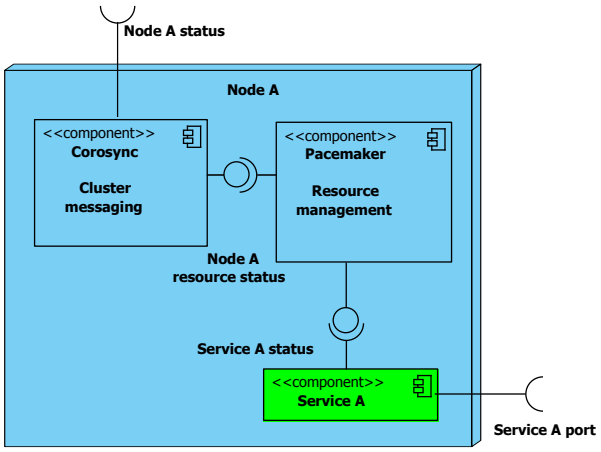

Figure 3.2 *Simple clusterized service deployment*

What kind of problems does this model have? The main problem is the interaction. To address it, section 3.1 can be used, and it will worth it to simultaneously draw the service interaction diagram. Service A wants to make a request to service B. How should it find where the service B is? As it was already discussed, in the single subnet environment, the virtual IP can be assigned to the service, and the requester should just query this IP. But this approach will give become harder to maintain for the mixed cloud infrastructure, since it will probably take to alter the routing table in the real time, assigning /32 subnets to each and every service. So the other way is to retaliate to more flexible approach used in SDN and based on the Directory

services. The directory will contain all the mapping, and the service can just pull the needed address from the directory (figure 3.3).

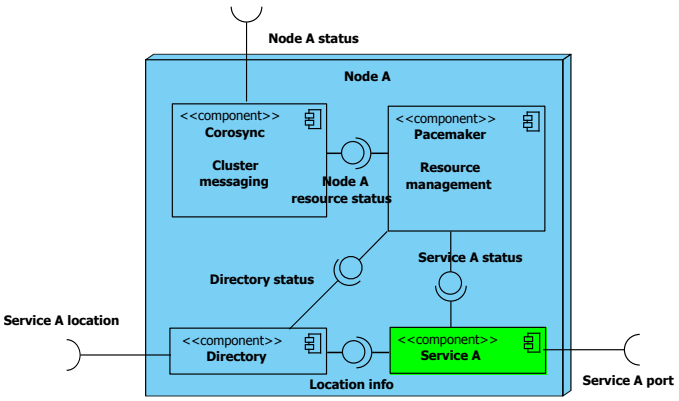

Figure 3.3 *Directory-based inter-service communication*

But this case violates the non-invasive principle declared above. The application should pull some data from the directory, which means that it should know about the directory in the first place, and it should know how to interact with this directory. So a proxy can be introduced. This proxy will take the directory interaction part, as well as the load-balancing. The service will now make a standard request to the local proxy, thinking that this proxy is a service B, and the proxy will redirect the request to the real service B (figure 3.4).

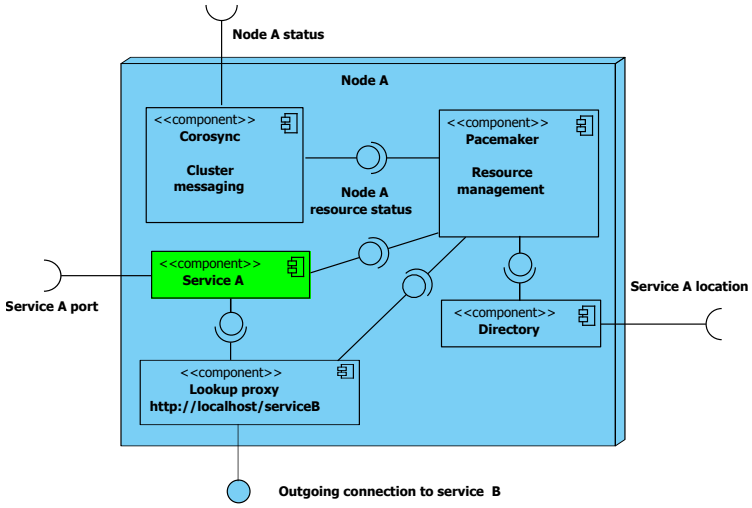

Figure 3.4 *Proxy-based inter-service communication*

The proxy should be configured. Some proxies, like Vulcand, can integrate with the directory out of the box, but the others, like HAproxy, cannot do this. Pros and cons of those two proxies were already discussed, so there is no need to start

this discussion from the scratch. Anyway, at some point some kind of real-time configuration service will be definitely required. For this, Confd can be used, which was originally made for Etcd as a part of the CoreOS project, but now it can work with almost any directory. This configuration service will act as an interpreter between the directory and the proxy config file (figure 3.5).

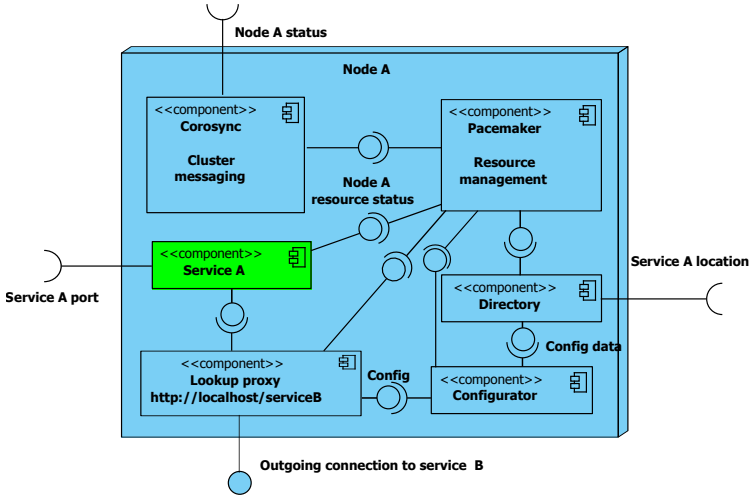

Figure 3.5 *Proxy runtime configuration*

Next comes the security. The service itself can provide no security at all, which is ok if it operates within the secluded environment. But when user tries to scale out, a simple Man-in-the-Middle attack can cause a lot of problems. The traffic should be end-to-end encrypted, which is commonly done using TLS and certificates. Moreover, to prevent the identity highjack the system should always check if the certificate is valid and trusted, and that it really is assigned to this particular requester. For this reason, a security gateway, or the direct proxy, can be implemented (figure 3.6). In this case, Nginx[11] was used, since it is fast and lightweight. And what is even better, it is scriptable.

This scriptability can give an additional benefit. Sometimes, the application itself can implement some kind of security model to separate the user roles. Tokens can be seen as one of the most popular solutions to make it possible. If some component of the application cannot use tokens, or if user wants to stop bogus requests even before they reach the component, the token checkup point can be implemented right on the security gateway. The service interaction diagram on figure 3.7 illustrates the complete login of server-to-server communication in this case. When one service

[11] https://nginx.org

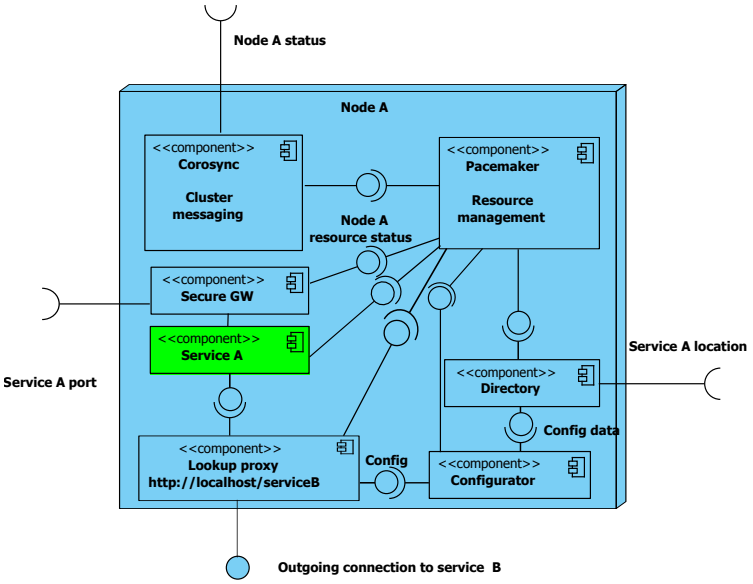

Figure 3.6 *Secure gateway as an entry point for the service*

tries to communicate to another, it first sends the request to the local reverse proxy, which will re-route it to the real service location. The request will land on the security gateway, or inbound proxy, which will perform minimal required security check before passing the request to the service.

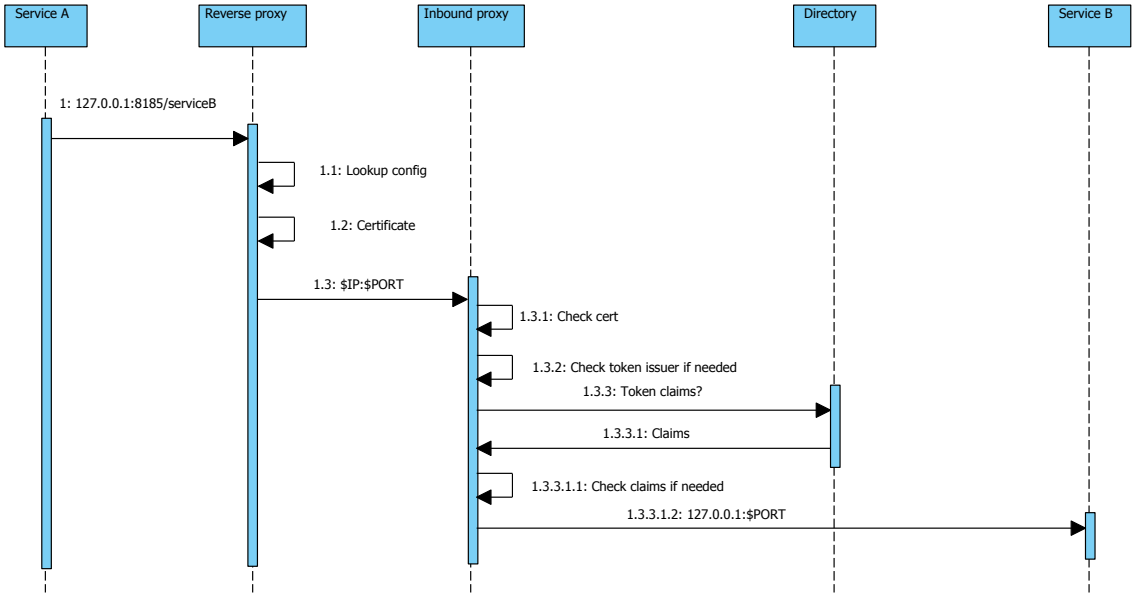

Figure 3.7 *Service-to-service interaction*

That is all for the interaction, but not for the components. There is one more case left - what will happen when the new component is introduced to the system, or

when the old component is moved or removed from it. This question is quite tricky, since no one knows what the abstract "component" should do. But the framework can make the user's life easier by adding at least some automation to the service discovery and registration process. For this, two more diagrams were made to show how exactly the user can interact with the system.

Nowadays, virtualization becomes more and more popular. One of the most popular microservice platforms is Docker[12], which claims to be the lightweight Linux container environment. The good thing about Docker is that it can publish some infrastructure-related events, such as the new container start event, through the socket. And listening to this socket is no different from listening to any other socket. So for dockerized services, to make the DevOps life a bit easier, the registrator service can be introduced. This service will listen to the Docker socket and populate the directory according to incoming events. The resulting setup is shown on figure 3.8.

From the DevOps' point of view, the process is explained on figure 3.9. In general, DevOps engineer should only prepare and build the Docker image. Most of the deployment-related actions are performed by the system itself, without DevOps involvement. Some actions, such as loading the image on each server, can be done in both manual and automatic way.

The process of dockerized service start-up is shown on figure 3.10. The registrator constantly listens to the Docker service, waiting for deploy/stop container events. When Pacemaker sends the command to deploy new container, registrator receives this event and publishes it into directory. Configuration manager, which listens to the directory events, populates the runtime configuration of all dependent services, such as proxies, based on this update.

The second way is to introduce some unified and easy-to-prepare service description model. This part will be described a bit later in 4.1, but even now the answer to this question can be given to illustrate the complete design after the last development iteration. In the end, Consul was used as an underlying directory, so its service model can be used to define custom, non-Docker services. The Consul use JSON-based service descriptions, which are easy to read and write. The process of introducing the custom service to the system in shown on figure 3.11. Instead of packaging

[12]https://www.docker.com/

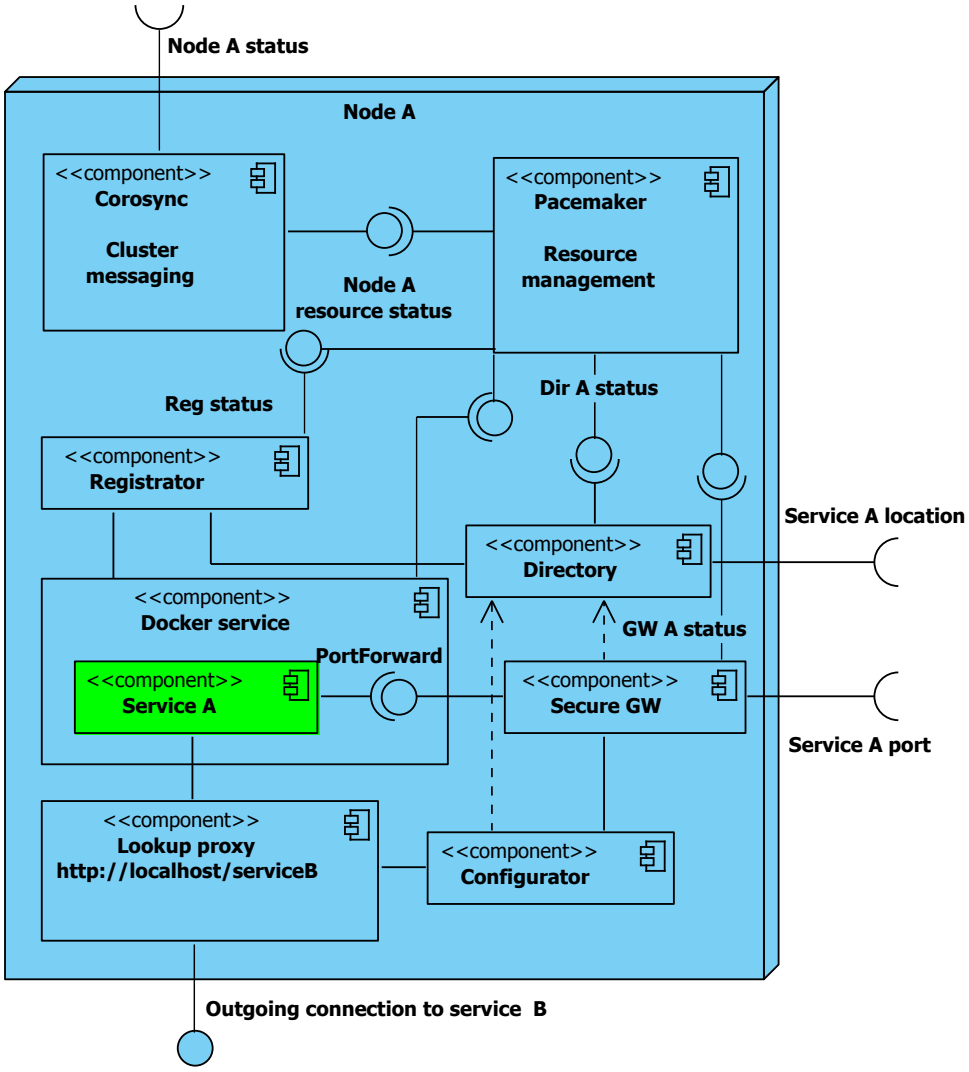

Figure 3.8 *Final system*

Docker container, in this case DevOps engineer will need to provide service health check and service description files, and give them to the deployer. Health check file is a standard Unix service tester, which should return 0 if the service works fine, and the description file is a simple JSON file of 5-10 lines long.

With this, all the design diagrams are finished. Next comes the implementation part.

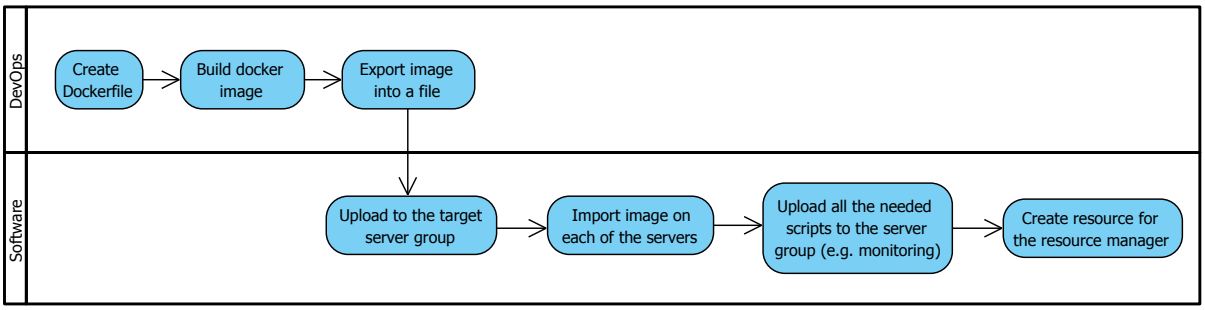

Figure 3.9 *Deployment process*

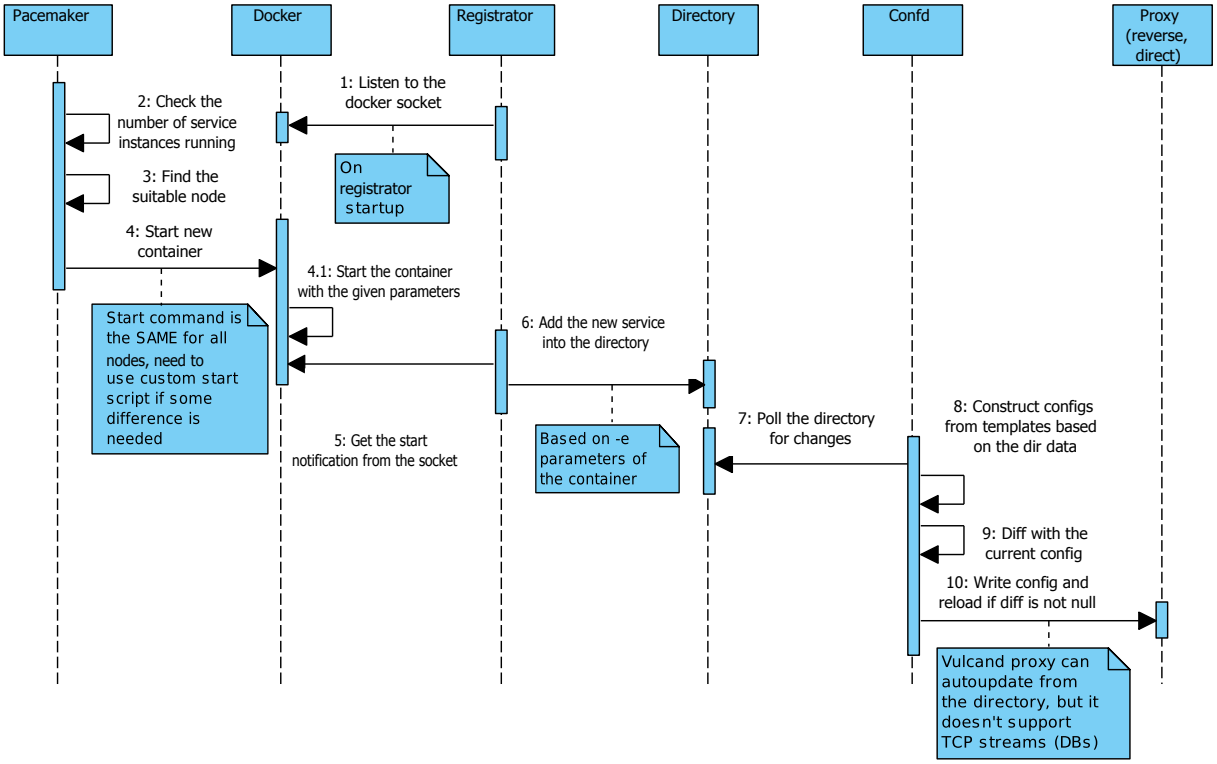

Figure 3.10 *New dockerized service startup process*

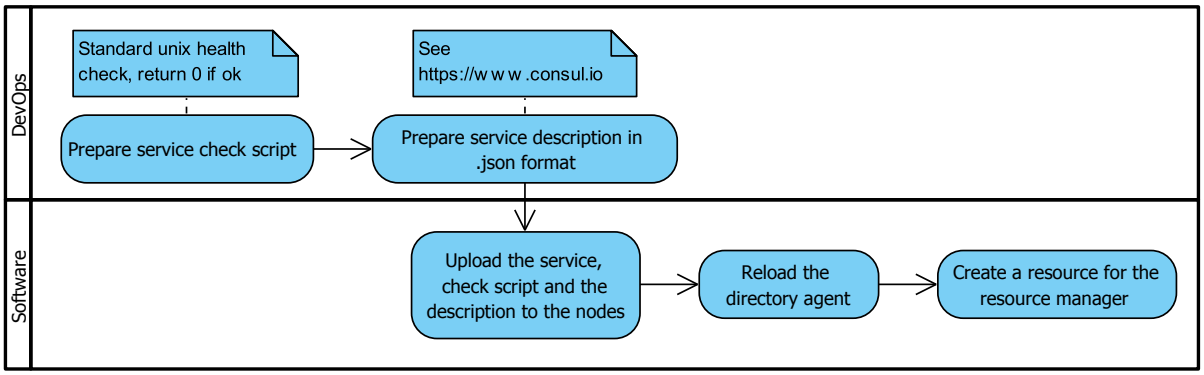

Figure 3.11 *Custom service deployment*

4. IMPLEMENTATION

4.1 Development iterations

The development process itself was performed using the iterative approach. In total, 6 iterations were made, containing different pieces of software with different configurations. The very first configuration included Corosync, Pacemaker, Etcd, Confd, Docker, Vulcand, and Node.js-based Etcd Registrator. Here, Corosync+Pacemaker suite was chosen as the most fast-developing and recommended Linux cluster management solution. Etcd was chosen for being lightweight and simplistic it terms of not being overloaded with additional functions that are not required for the framework functionality. Confd was taken as the recommended runtime configuration manager for Etcd, since both of them are developed in the CoreOS project scope. Docker is nowadays the most popular container environment for microservices. Vulcand was chosen as a load-balancer and a reverse proxy because of its runtime reconfiguration features and close integration with Etcd. Node.js-based Etcd Registrator was taken because it was capable of writing the directory structure compatible with Vulcand without additional restarts.

The second iteration switched from Vulcand to HAproxy, which is capable of managing TCP connections. TCP connections are crucial for databases, and most of the non-web-services, since in many cases HTTP wrapper use is either impossible or pose to much of overhead.

The third, fourth, and fifth iterations were dedicated to the Access Control List (ACL) management. At first, Tyk[1] was implemented as a direct proxy to check the ACL for the particular endpoint. The main problem of this configuration was the use of Redis[2]. Redis is a very fast key/value storage, but its clustering mechanism is still a bit underdeveloped. Redis documentation[3] for clustering states that the

[1] https://tyk.io/
[2] http://redis.io/
[3] http://redis.io/topics/cluster-tutorial#creating-and-using-a-redis-cluster

minimal condition for building the cluster is having 3 master nodes. Adding the master-slave pair requirement imposed by Redis' HA mechanism[4], it sums up to the minimum amount of 6 nodes inside the cluster, plus one more node as by the rule of thumb cluster should have odd number of nodes to use the majority rule effectively. Having seven nodes can be a tough requirement for a small-scale system, thus the decision not to use Redis was made.

Next attempt was to use Kong[5] ACL manager, which relies on Apache Cassandra[6] key/value storage as a backend. Cassandra was originally developed to run as a cluster, and once set up correctly, it is very hard to crash the cluster completely. But it turned out that Cassandra has one big problem when used in virtual environment. Even the empty instance during the startup consumed 0.5-1 Gb RAM, which means that not every VM will be able to run it. During the test run, Cassandra's memory consumption peaked at 2 Gb, making it completely unacceptable as a VM-based microservice backend.

The fifth iteration was to use Nginx together with the custom Lua script to check ACLs stored in already-deployed directory. This method proved to be the most flexible and lightweight. As Lua scripting support is not included in most of the Nginx distributions, a custom build OpenResty was used. The script itself is quite simple, it makes use of *lua-resty-jwt* and *lua-resty-consul* libraries. First, it looks up if the node '*/acl/$service*' exists in the directory. Next, if the node exists, it checks if the JSON Web Token (JWT) is present in the request header. If the token is present, it tries to decrypt it and verify claims, otherwise it returns code 403. If node is not present, the default behavior will be triggered.

The last sixth iteration was dedicated to the directory interaction improvement. Etcd works well, but it takes a considerable amount of work to add a custom service definition. Thus, Etcd was switched with Consul alongside with the different registrator service and the different runtime configurator. Consul provides simple custom service definitions based on the JSON description files. It also allows to implement service health checks in much easier way. This service-centric approach was also a reason to switch from Confd runtime configurator to Consul_template.

After the sixth iteration, the overall software collection is the following: Corosync,

[4]http://redis.io/topics/sentinel
[5]https://getkong.org/
[6]http://cassandra.apache.org/

Pacemaker, Consul, Docker, HAproxy, Nginx (with Lua scripts), Consul_template, Gliderlabs Registrator[7].

4.2 Development process

Initially, it started from creation of the document, which contained the planned software list and the deployment script table. As originally it was not clear which deployment model will be used, this table contained a simple shell code to perform each step of the deployment.

The first step was to install and configure Corosync+Pacemaker combo. This combo allows to create a simple standalone cluster in a way it is usually used in LAN cluster environments. The deployment script for this step included the automated config file creation based on the '/etc/hosts' file contents and the cluster hosts naming convention, which was assumed to be 'clusterhost[\d]' at that stage.

Next, the directory was added to the stack to provide replicated key/value storage capabilities. Once again, the sample configuration file creation was written in the document alongside with the comments on some typical configuration issues. After the configuration file creation, the directory service was passed to Corosync+Pacemaker stack for management.

The next step was to deploy runtime configuration management. Originally, no additional templates were written, so only the deployment and configuration was handled. Both Confd and Consul_template are provided as-is, without any system service definitions or packages. So, the deployment script was not only downloading the software itself, but also writing a service definition and a Pacemaker OCF script.

Next piece of software to be added is the Docker service. As it is quite popular, and it is included as a package in many mainstream distros, the script just pulls the package and adds the service to Pacemaker for management.

Last three parts added are the reverse proxy, the direct proxy and the registrator. For those three, the installation process was not that straightforward, since these pieces of software are distributed either as a source code, or as a docker image. Source code needs to be compiled, and docker adds significant overhead to the operation.

[7] http://gliderlabs.com/registrator

The way that was found was to use docker to compile binary with the host system parameters, and then to use this binary on the host system, removing the docker container after the compilation. Additional step might be to wrap the binary into the package to pass the control over the installation to the package manager, this proposal is discussed more in chapter 7.

4.3 Deployment

Deployment scripts were written in Chef. Chef is a mature, agile deployment system made as a number of Ruby libraries. It means that the deployment script itself, called "the cookbook", is a Ruby script which makes use of those libraries. The cookbook itself can refer other cookbooks, include recipes and libraries from those. In total, the main cookbook refers to over 20 additional cookbooks. Some of those were used as-is, some were patched, and some were created from scratch and published as a separate open source projects. The full list of contributions made while working on the framework is provided in section 4.5 of this chapter.

Each cookbook created for this thesis includes a number of unit tests, embracing the test-driven development principle. For those tests, the specialized flavor of the rspec utility, the chefspec, is used. Those tests check if the main recipe can converge, if it includes all the needed recipes, and if those recipes parse the provided options correctly. In addition to chefspec, rubocop gem was used to as a linter (code style checker), and foodcritic gem was used to ensure that the recipe follows the Chef recipe design conventions. Also, since all gems listed above check only the recipe but not the real workflow, the FAST Lab's GitLab CI continuous integration platform was used to perform live integration tests.

The main cookbook includes the main recipe and a number of additional recipes, one for each component installed. Those additional recipes are called by the main recipe in the particular order, starting from the cluster messaging. Initial parameters are provided as a JSON file. The minimum configuration includes the node list and the Pacemaker node authentication token. It is also recommended to provide the Corosync cluster key file, which can be generated using any random number generator.

To start deploying the framework, user should first designate one of the machines as a Chef server. It can be one of the cluster machines, but it is not mandatory. The

only requirement is that this machine's Chef server port should be accessible by all future cluster members, and the machine's FQDN should be resolvable. After the Chef server is configured, the recipe should be downloaded to the server's cookbooks directory. Next, using the Berkshelf Ruby gem, download all the cookbooks require by the recipe. Then, use the chef-ssl gem to create an SSL CA, if you want Chef to generate certificates on its own. At this point, the server configuration is finished, and the bootstrap command for each node can be issued. After the first run, use the chef-ssl gem to generate SSL certificates and run the chef-client once again on each of the nodes using 'chef ssh' command from the server. The second run will be much faster since it does not reinstall anything.

On some distros the version of Ruby interpreter might be too old to use chef-ssl. The solution is to use Ruby Version Manager, or rvm. It allows to install the newest version of Ruby into a separate directory.

The full installation from scratch takes about 17 minutes on virtual machine, which use a single core of an Intel Core i5 CPU. Most of this time is consumed by the registrator and the direct proxy installation, since those are compiled from the source and not installed from the package. In case of the registrator, source installation is not necessary, but it saves a lot of resources, since the other deployment possibility is the Docker-based deployment. For the direct proxy, the source install is required to enable Lua scripting support module, which is used to perform JWT-based security checks.

4.4 Configuration

Configuration includes the preparation of the direct and reverse proxy config templates. Those templates are used by the configuration manager to make the system fly. The templates are written in Go Template language, which is simple yet quite ugly. These templates are provided in appendix A.

The other part of the configuration is the JWT check script for the direct proxy. This script is written in Lua, its operation sequence diagram is shown on the figure 4.1. The script is integrated with the directory, from which it takes the list of claims needed to be present in the JWT to grant access.

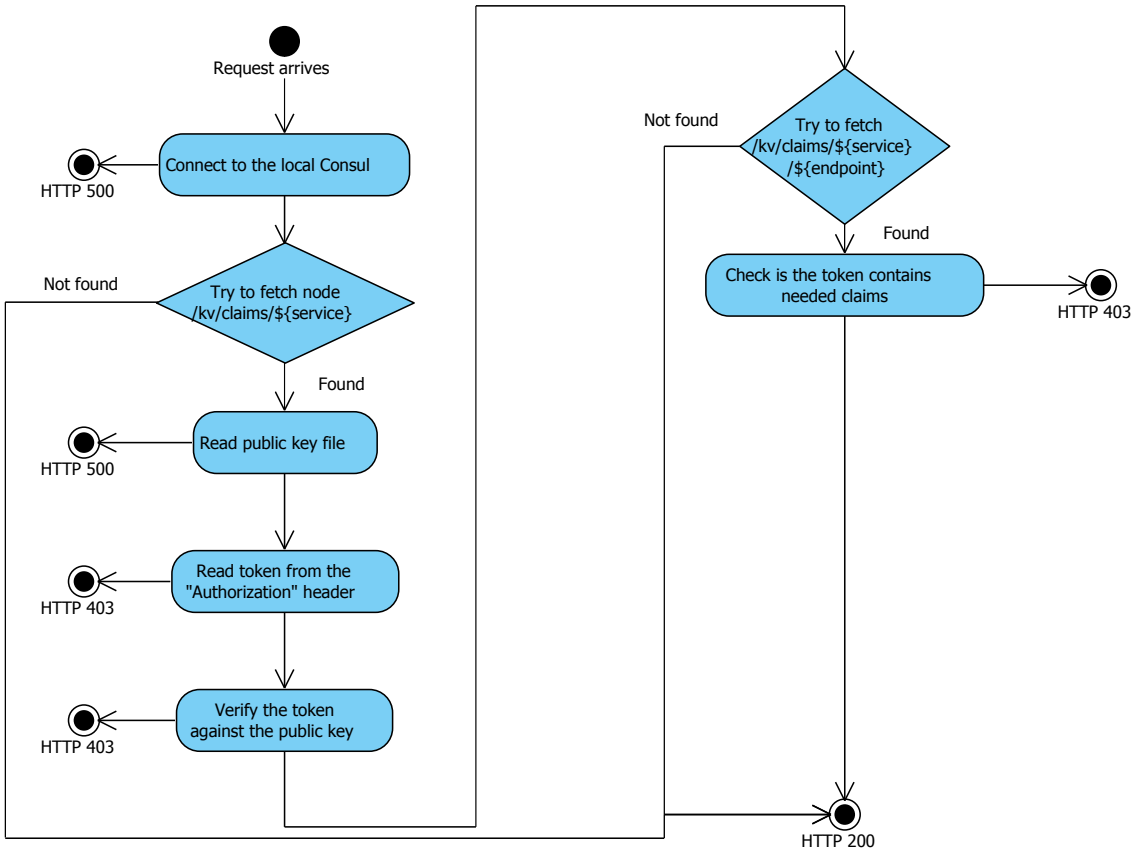

Figure 4.1 *JWT check algorithm*

4.5 Patches and contributions

In the scope of this thesis, a number of third-party projects was used. In many cases, those projects were lacking some functionality needed to bind them together. Since Free and Open Source Software (FOSS) philosophy encourages the "give and take" principle, a number of patches was created and sent to the upstream developers, and some pieces of software were created from scratch and published under the FOSS licenses. The following list summarizes the contributions made during this thesis work.

- github.com/psi-4ward/docker-etcd-registrator: 4 patches accepted, became a colaborator
- github.com/vulcand/vucand: 2 patches to be accepted
- github.com/bellpeterm/eassl3: 1 patch accepted

- github.com/rtkwlf/chef-x509: 1 patch accepted
- github.com/target/pacemaker-cookbook: 1 patch to be accepted
- github.com/adamkrone/chef-consul-template: 1 patch to be accepted
- github.com/chef-cookbooks/etcd: 1 patch to be accepted
- github.com/priestjim/chef-openresty: 1 patch/issue accepted
- github.com/upcfrost/chef_corosync_cookbook: created, published on Chef Supermarket
- github.com/upcfrost/gliderlabs_registrator: created, published on Chef Supermarket
- github.com/upcfrost/docker-etcd-registrator-cookbook: created, published on Chef Supermarket
- github.com/upcfrost/chef_vulcand_cookbook: created, published on Chef Supermarket

The final general cookbook for the HA framework, with all scripts and configuration files, is not yet published, but it is highly possible that it would be also published as an open-source software, and also as a part of MUSA project.

5. RESULTS

5.1 User-side overview

The framework grants the following possibilities to the user:

- Fully automated deployment of containerized applications. In this scenario, the only action required from the user is to specify the new application resource. HA capabilities, automated deployment and routing will be performed automatically.
- Replicated key-value storage with REST interface that can store runtime data accessible for all local apps.
- Automated configuration of the services to decrease reconfiguration time before redeployment.
- High availability and load balancing mechanisms with location-agnostic routing.
- Additional encryption and configurable security checks for web services to spare user from in-source SSL configuration.
- Mechanism to control running services from any node within the cluster.

All of those possibilities are non-invasive, and some of them, like KV-storage, provide additional features in case developer will decide to use them inside the application. All of those features can be configured either from Web Graphical User Interface (GUI), or from the Command Line Interface (CLI). The framework provides two main interfaces for the user: clustering interface and directory interface.

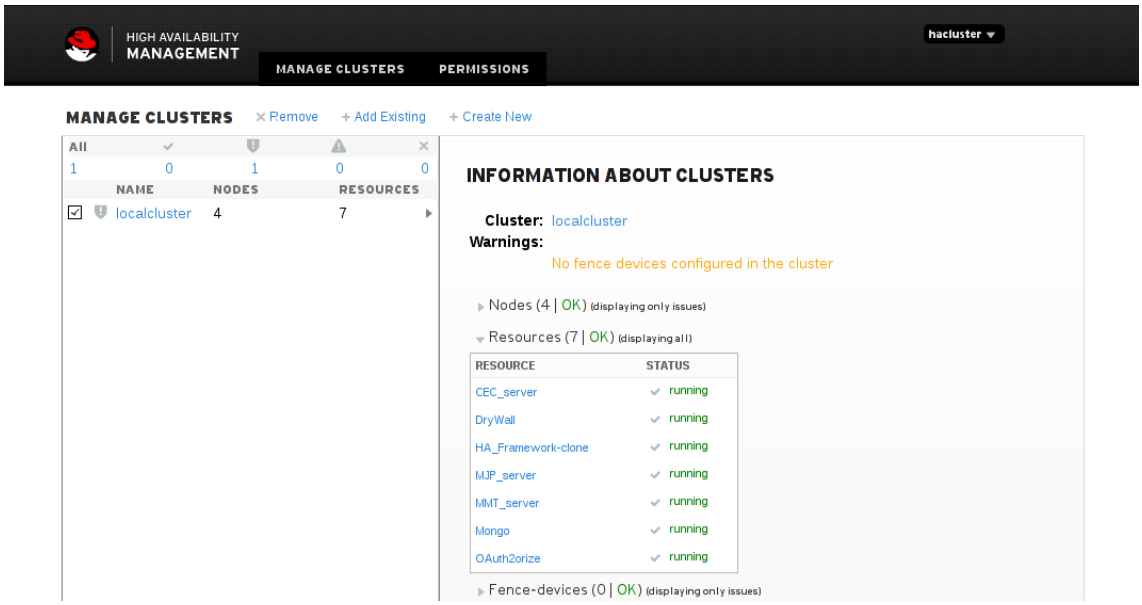

Figure 5.1 *Clustering Web GUI main screen*

5.1.1 Clustering interface

Clustering interface is available in two different versions: CLI and Web GUI. Both versions are available from any fully operation node. If changes are performed on non-operational node, these changes will be discarded when the node will join the cluster. Clustering Web GUI main screen is shown on picture 5.1. Web GUI is accessible over HTTPS protocol at port 2224 on any node inside the cluster.

Clustering interface, which controls Pacemaker cluster manager, allows OS-level and application-level management of the system. The current cluster configuration is stored in so-called Cluster Information Base (CIB), which is a replicated XML-style configuration file. CIB stores node data, cluster runtime configuration, and resources configuration. Resource is a generalized entity managed by the Pacemaker, it can represent a service, a script, a filesystem, or any other manageable entity. Pacemaker gets information about how to manage this particular resource from the agent description file. In fact, resource is an instance of the agent. The information the agent usually provides is at least how to start, how to stop, and how to monitor the service. Agents come in two different types - LSB agents, or the standard Linux services, and OCF agents, made as custom shell scripts. It is hard to say which type is better. LSB agents have better OS integration, but they are distro-specific and usually not very flexible in case when systemd is used. OCF agents are more flexible, but they do not integration with the underlying OS well, and usually they

are custom-made.

Each agent can take a number of parameters. The most common ones are monitoring options, e.g. how frequently the cluster should request the health report from the resource. LSB resources are usually configured in the same was as they would be configured for the standalone use, for example using the configuration files. OCF resources can handle a number of resource-specific options, like the database location or the service port number. These parameters are replicated across the cluster, but it is possible to make some of the parameters node-specific. Since in this thesis, as well as in MUSA project, multicloud applications are largely considered, one of the important parameters of any service is the resource-stickiness value, which literally means that the service will stay at the node with the highest stickiness unless this node goes down. It is important since in multicloud environment user might want to restrict some of the services to particular nodes.

Comparing CLI with Web GUI, CLI is more flexible and more stable, it allows some configurations not supported by Web GUI, and it makes possible the direct CIB modification. Web GUI is more user-friendly, and it has a good multicluster overview screen. Many big distros provide their own Web GUI for Pacemaker, e.g. SUSE provides Hawk interface. In this thesis, Red Hat's PCS interface was used. PCS provides not only the Web GUI itself, but also a number of REST endpoints, making it easy to create a custom GUI.

Next follows the short list of examples on how both CLI and Web GUI can be used.

- To add a new resource, the following steps can be followed. These steps correspond to different capabilities of the framework, so none of them is strictly required. The framework is most optimized when used with dockerized web-services. To add such service, just add a new resource with type "docker", set the container network mode to "bridge", specify the name of the service similar to the application base URL (e.g. "app1' for /app1), and open the needed ports.
  For docker, the example command (listing 5.1) to add a new resource might look as the following. The environment variable SERVICE_80_NAME will be used as a base URL, which in this example will be "/phptest".

Listing 5.1 *Example command to deploy a new service*

```
pcs resource create TestPHP \
 ocf:heartbeat:docker reuse=false \
 image="richarvey/nginx-php-fpm" \
 run_opts="--net=bridge -p 127.0.0.1::80 \
  -e SERVICE_80_NAME=phptest \
  -e SERVICE_80_TAGS=http \
  -v /home/test/tmp:/var/www/html"
```

The full workflow for resource of any type is the following. First, if you would like to use automatic resource management and fail-over, find or create the corresponding resource agent. The list of pre-made resources can be found at Linux-HA website [1]. Additionally, any systemd-controllable service can be introduced as a resource to the system. The main drawback of this way is the monitoring capabilities limitation, as systemd simple unit test format cannot handle complex sanity checks, which will still require custom shell script to be written. The other way is to write a custom OCF script using the dummy example available at Linux-HA website. Next, create a resource using this agent and specify the nodes where it should run. After you will check that the resource runs, add a service description file to each node where the service should be able to run if you want to provide automatic endpoint mapping. The example of the service description file can be found at Consul website [2]. In tags specify the endpoint type (HTTP/TCP) and if automapping is needed. If the custom application health check script is needed (e.g. if systemd agent resource is used), please write the simple health check shell script which should return 0 if service works correctly, and any value greater than zero otherwise. Next, add the endpoint address (port and/or URL) to the Consul K/V storage.

- To add new node to the cluster in a manual way, the first step is to provide a common pre-shared key and a configuration file to the new node. Both key and file can be taken from any other node. Next step is to add the new node to all existing nodes. It can be done faster by adding the new node to the configuration file of one node, and then resyncing the configuration from this node to the cluster. The last step is to start the cluster service on the new node.

[1] Linux-HA website: http://www.linux-ha.org/wiki/Resource_agents
[2] Consul website: https://www.consul.io/

Both CLI and Web GUI allow to skip most of these steps, adding the new node with one simple command. There is yet another way to inject a new node - the data can be injected directly into CIB, which allows to add and remove nodes at runtime without handling the config file. The drawback is that the cluster should persist all the time, otherwise it will revert back to the saved configuration.

- To stop one of the nodes for maintenance, from the CLI issue the following command

  ```
  pcs cluster stop ${nodename}
  ```

  In Web GUI, open the cluster of interest, go to the "Nodes" screen, select a node, and click the "Stop" button.

- To restart one of the resources, from the CLI issue the following command

  ```
  pcs resource restart ${resourceName}
  ```

  In Web GUI, open the cluster of interest, go to the "Resources" screen, select a resource, and click the "Disable" and "Enable" buttons.

- To reconfigure one of the resources, from the CLI issue the following command

  ```
  pcs resource update ${resourceName} ${newConfig}
  ```

  In Web GUI, open the cluster of interest, go to the "Resources" screen, select a resource, reconfigure, and click "Apply changes" button.

5.1.2 Directory interface

Web-based directory interface provides a limited access to the Consul directory service. It allows monitoring of the Consul cluster state, showing status of each service registered in the directory, both autoregistered and custom. It also gives access to the key-value storage interface, which is by the framework to manage ACLs. The overview of the WebUI interface is shown on figure 5.2. Directory Web UI is accessible on port 8500 at any node at */ui* endpoint..

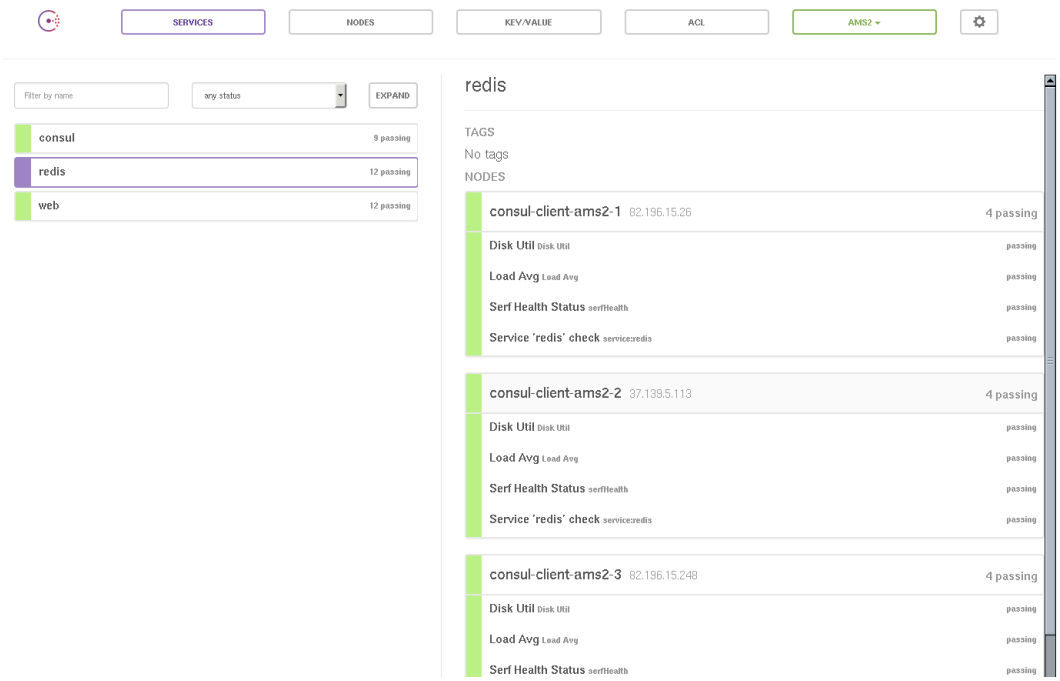

Figure 5.2 *Directory Web UI (Source: http://demo.consul.io)*

In addition to the web interface, it is possible to interact with the directory using REST requests. The REST interface is accessible to any REST client making request to the port on which Consul is listening (default: 8500). Internal ACLs can be used to restrict access to the directory. An example REST query is shown on listing 5.2. This query will return the brief info about the service itself, the nodes where it runs, and health check results for each instance.

Listing 5.2 *Consul REST query example*

```
http://127.0.0.1:8500/v1/health/service/test
```

RESTinterface can be also used to add data into the directory. In most cases it is done by using HTTP PUT requests, with the request body containing instructions in JSON format. It allows to use the directory as a cache, as well as to create custom configuration templates to reconfigure services upon user-chosen event.

5.2 Testing

As it was stated before, main capabilities of the HA framework are:

- Location-agnostic routing

- Load-balancing
- Automated fail-over and resource management
- Traffic encryption and API access control

This chapter summarizes various tests performed to evaluate these capabilities.

5.2.1 Setup description

For testing purposes, one of the virtual machines used for the MUSA project was used. Those machines are provided by TUT. Virtual hardware-wise, the machine has 1 CPU core, 2 GB RAM, and 14 GB HDD. Software-wise, the machine runs CentOS 7 in basic configuration, with the latest updates for August 2016. The HA framework was installed on all MUSA project machines beforehand.

5.2.2 Tests and results

Various tests were performed directly and indirectly, including network bandwidth consumption, memory and CPU consumption, and fault-tolerance.

Based on the iftop utility output, the framework itself requires about 40 kbit/s bandwidth to synchronize 4 nodes. About 90% of this bandwidth is generated by Consul to keep its directory intact. This bandwidth consumption across the multicloud installation can be optimized by grouping nodes into virtual data centers[3]. Inside the datacenter the bandwidth consumption should not be critical.

Memory-wise, the framework takes about 150-200 Mb RAM, plus approximately 50-70 Mb RAM for the OS itself, which makes it possible to use in low-powered VM environment. Main memory consumers are Consul, Corosync, and Docker. Corosync reports quite high memory consumption of 200 Mb, but in fact it takes much less, about 50 Mb, using the rest as a pre-allocated buffer. This memory can be still used by other software. Pacemaker web GUI can be turned off if needed, it consumes about 50 Mb of RAM. Docker can be a memory hog, and it is a known fact that

[3]Consul datacenters: https://www.consul.io/docs/guides/datacenters.html

Element	CPU consumption	Memory consumption	Traffic per second
Corosync	30 MHz	50 Mb	2 kbit/s * (N-1)
Pacemaker	30 MHz idle	50 Mb	-
Consul	50 MHz	50 Mb	10 kbit/s * (N-1)
HAproxy	-	10 Mb	0.5 kbit/s * S
Nginx	-	10 Mb	-
Registrator	-	10 Mb	-
Docker	Service-based	20 Mb (empty)	-

Table 5.1 *Approximate idle resource consumption summary*

container engine always have quite large memory footprint. Docker takes about 50-100 Mb RAM. With additional tweaks the framework's memory consumption can be reduced to approximately 70-100 Mb.

CPU-wise, in the original setup the main CPU consumer was Etcd running in SSL mode, since its encryption routines are not well-optimized. With Etcd on the current setup the CPU queue peaked at 30%. After migration to Consul this problem was solved. Still, Consul and Corosync are two main CPU consumers as they constantly use encrypted messaging, but the overall CPU consumption is quite low. Consul might produce larger CPU load in some cases, but usually it takes significant amount of read-write access to overload it.

The final idle resource consumption summary is shown in table 5.1. Here, N is a number of nodes in the cluster, and S is a number of service instances. The CPU consumption was estimated using a single-core 1 GHz virtual machine. Values provided in this table are approximate and heavily depend on the amount and nature of services deployed on top of the framework.

To test routing and load-balancing capabilities, a simple test was performed. Sample service with a simple REST-based counter was deployed on two different nodes. The service was deployed as a Docker container using the *nginx-php-fpm* base image[4]. The counter code is provided in appendix C. The service was deployed using the command provided in chapter 5.A number of requests was poured onto the cluster through the external gateway. After the test, the counter values were checked. The total number of requests was 1000. To send the all the requests, the following shell command was used. All the requests were sent to the reverse proxy without explicit definition of the receiving server (listing 5.3).

[4]Image URL: https://hub.docker.com/r/richarvey/nginx-php-fpm/

Listing 5.3 *Request loop for the sample service*

```
for ((i=1; i<=1000; i++)); do
  curl http://127.0.0.1:8185/phptest/index.php;
done
```

As a result, both nodes received exactly 500 each. It means that both routing and soft load balancing work fine.

To check the fail-over capabilities, the following test was performed multiple times. Two "counter" services from the previous test were launched on two different machines. After the request loop started, one of the machines was shut down. From the output it was seen that twice the server reply was incorrect. Those two bad response bundles were generated first by the service shutting down as there was no special instruction to the HAproxy how to handle responses other than HTTP 200, and the second one by the HAproxy renewing its configuration. The configuration renewal process takes less than a second, but since requests were sent constantly, some of those still managed to hit during the reload time. In total, during the typical test, the first node received 3153 requests, the one being rebooted processed 175 requests, and the third node, to which the service migrated, handled 1632 requests, with 4960 out of 5000 total requests processed in average. As one request takes about 26ms measured by the 'time' command, it means that the total downtime of the system was about 1 second. In fact, it was lower as approximately 30 requests were rejected almost immediately without any delay during the HAproxy reconfiguration, which takes less than one second. It can be further reduced by introducing graceful reload of the HAproxy, and adding instructions about how to proceed in case of different HTTP response codes from the upstream server. With only the HAproxy reload tweak it is possible to correctly process on average 4990 requests out of 5000.

Next feature tested was the ACL based security. First, the JWT public key was put into the '*/etc/nginx/lua/certificate*' file. In this test, a sample public key from the `jwt.io` page was used. The certificate file should be properly formatted, 64 characters per line. Then, the JWT check procedure for this service should be enabled. To do this, the directory record that corresponds to the service name should exist (listing 5.4).

Listing 5.4 *Example code to enable JWT certificate validation*

```
curl -X PUT -d 'test' \
```

```
http://localhost:8500/v1/kv/claims/phptest
```

After this, a simple authorizationless request to the service will return the HTTP 403 response. To get the page, a valid token in the "*Authorization*" header field should be provided. One again, the default token from the `jwt.io` page was used. After providing the correct token, the JWT check gives green light to the request. Now let us add some more meaningful claim to be checked. For example, '/index.php' page requires 'ROLE_ADMIN' claim set. First, let us add this rule into the directory (listing 5.5).

Listing 5.5 *Example code to add the ACL claim check rule*

```
curl -X PUT -d 'ROLE_ADMIN' \
  http://localhost:8500/v1/kv/claims/phptest/index.php
```

Now the script will check if the '*ROLE_ADMIN*' is specified within the '*authorities*' array of the JWT. The sample token which provides the correct value is shown below in listing 5.6. This token structure is acknowledged by many security frameworks, such as Java Spring Security framework, but if the custom JWT structure is needed, the checkup script can be expanded manually.

Listing 5.6 *Example token*

```
{
    "id": "e0ad1ef3-a8a5-4eef-998d-00b26bc2c53f",
    "user_name": "john",
    "exp": 1458126622,
    "authorities": [
        "ROLE_USER",
        "ROLE_ADMIN"
    ]
}
```

The last feature tested was the communication time overhead from the framework. In this test, both inbound gateway and the service were placed on the same node to neglect the network latency. To track the request progress, tcpdump was used to sniff the traffic. The dump was later decrypted using Wireshark. In this dump, a few notable points-of-interaction can be seen. One of such points is presented in table 5.2, which was made using one of the Wireshark dumps. This table shows

the path of the request coming from the outside world, which also includes some overhead from the gateway. Packet 179 arrives to the gateway requesting the service. Packet 182 is the same request redirected by the gateway to the reverse proxy. The communication between the reverse proxy and the inbound proxy is not visible as it is encrypted. This is the only external connection between the servers, thus the only one that can be intercepted if the server itself is not compromised. Packet 195 is sent to the directory by the inbound gateway ACL check script, meaning that at this point the request has already arrived to the destination node. The request finally arrives to the service at with packet 216, meaning that the total time overhead produced by the framework is around 7 ms. Using the diagram 3.7, the timing from the table 5.2 can be visualized (figure 5.3).

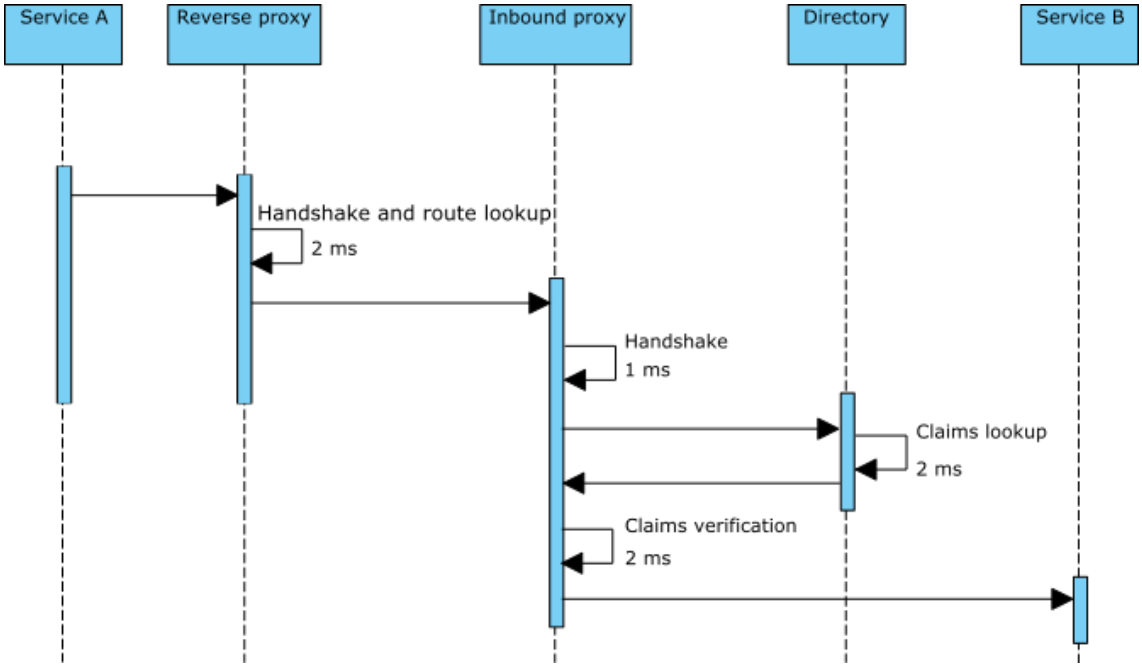

Figure 5.3 *Service timing visualization*

#	Time	Source	Destination	Protocol	Description
179	1.290987	Requester	HAproxy A	HTTP	Service request
182	1.291072	HAproxy A	Nginx B	HTTPS	Inter-server request
195	1.294138	Nginx B	Consul B	HTTP	Claims check request
203	1.296003	Consul B	Nginx B	HTTP	Claims check response
208	1.296832	Nginx B	Docker B	HTTP	GW to Docker
216	1.297635	Docker B	Service	HTTP	Docker to service

Table 5.2 *Inbound request progress*

By re-running the same process with different queries it can be seen that the overhead

is not getting greater than 10 ms per internal routing operation, and it gets lower for the following requests because of route caching. One of those 7 ms, approximately 3 ms are spent on the TCP handshakes between the components, and 3 ms more are added by the docker proxy. Docker is known to add communications overhead, and not much can be done here. The TCP handshake overhead can be decreased in many cases by using long running TCP connections. For HTTP, it can be done by switching the communication between HAproxy and Nginx to HTTP/2. Still, it will only allow to drop the longest handshake that possibly goes between physical servers. Shorter handshakes, such as direct-proxy-to-directory and direct-proxy-to-docker, cannot be dropped without underlying software modification, including the service hosted inside the docker container.

6. DISCUSSION

The resulting system includes the following components:

- Corosync as a clustering mechanism provider.
- Pacemaker as a resource manager. It monitors the resources and tracks how many instances are currently running. Pacemaker also handles migration of the resources in case of failure.
- Consul as a directory backend. It stores runtime configuration of the system, such as routing information, and provides some additional monitoring mechanisms.
- HAproxy as a reverse proxy and load balancer. It employs the data stored in Consul to provide agnostic routing to the requesting applications. It also includes some simple health checks and provides HTTP-to-HTTPS tunneling using certificates.
- Nginx that acts as an inbound secure gateway. It provides HTTPS certificate checkup and tunnels the HTTPS request to the application HTTP endpoint. It is also capable of running additional checks using Lua scripts.
- Docker service to allow simplified deployment of the containerized software.
- Gliderlabs Registrator that listens on the Docker socket and populates the Consul based on the incoming data.
- Consul_template as a runtime configuration manager, that prepares configuration templates for both HAproxy and Nginx based on the Consul contents.

One of the main points of the resulting system is described by the word “framework” used in its name. Most of these components can be used separately, e.g. Consul can

be used to store custom user data, and HAproxy can forward requests to the hosts outside of the cluster. The use of Docker also does not force user to use the whole framework.

Compared to the normal cloud infrastructure, the main benefit of using the HA framework is the ability to mix the public cloud with local on-premises infrastructure while having a single control point over the whole cluster. It also simplifies the routing and migration between the locations, allowing to move from the local infrastructure to the cloud and vise versa in case of emergency. The price for this flexibility is higher latency as service part instances might run in different locations. As the choice of instance location can be enforced on the resource manager side, careful configuration can help in mitigating this issue.

When comparing to specialized solutions, the HA framework's main benefit is the flexibility in terms of software and hardware. Of course this flexibility does not come free, and both monitoring and performance capabilities of the framework are lower compared to the specialized systems.

Compared to the container-based solutions, the framework is capable of running custom non-dockerized services. Docker can impose significant overhead on the real-time services, such as telephony and fax. The drawback is the ease of migration. In container-based solutions, the migration process is very simple - the container is stopped on one node, uploaded to another, and started there. If shared network storage is used, there upload step can be skipped, making migration process even faster. The paused container can also keep the runtime state of the software, while in the described solution this state should be saved explicitly if needed. But in fact using paused containers in this way is considered to be a bad practice.

In terms of High Availability, the framework provides quite good performance, limiting the amount of failed requests to one at maximum. This one request can get stuck if it arrives in between the check requests in case of the service failure and in case of reverse proxy misconfiguration, for example if no repeat is configured or service response waiting timeout is too high. In default configuration, every request should get the response in case if at least one instance of the service is healthy. If only one instance of the service is running across the whole cluster, the availability might be worse. First of all, the reverse proxy will mark the failed service and, as there are no backends left, it will reject the request. Second, the service startup time can be quite long, especially for some heavy applications. Third, if the docker

image was not originally provided to the target node, the pull time can get very long, up to few minutes.

In terms of general performance, amount of resources consumed by the framework is quite low. In a way, it would be even possible to run the HA framework-backed services on single-board ARM machines like Raspberry Pi. The network overhead is also quite low, though 10 ms per hop means that the number of hops should be kept low. Of course it is possible to create a long chain of inter-server requests, which will result in delay of hundreds of ms, but in this case the network latency between physical machines will create greater problems, as the multicloud nature of the overall setup should be always taken into account.

One positive point about this thesis work in general is the amount of public contributions, which, in a way, ensures that at least some parts of the HA framework will be definitely used by the general public. Some of the patches were made not only because they were needed for the framework to work, but also because they were registered as an issue on the official component's bug-tracker.

Currently the framework has two main problems. First one is the firewall. Some cookbooks require additional data to be downloaded from various sources, including external git repos. Some of those repos are using git+ssh, which is usually blocked by most of the corporate firewalls. The second big problem is the external source stability. At the moment when this sentence was written, `luarocks.org` was down, meaning that additional configuration was required to install the framework. One way to deal with those problems is to create a bundled installer that will contain all the software needed for the framework. This way is the most beneficial in terms of control and updates, but it might take a lot of additional support work because of different Linux distributions available at the market. The other way is to create a local mirror with the required software and to redirect requests to this mirror. This way is quite hard to implement as it will take a lot of scripting with a very small outcome.

In the real life, the framework is used as an underlying platform for the MUSA project[1]. The setup consists of 5 nodes. Four of these nodes were hosted at TUT, the fifth node was hosted at AIMES[2] private cloud in UK. The setup itself mirrors the standard mixed cloud setup, with a number of on-premises servers and a few

[1] MUSA, EU Horizon 2020 project: http://musa-project.eu/

[2] www.aimes.uk

outsources machines. AIMES was chosen since they have a certificate required to handle medical data. See table 6.1 for machine-wise setup description.

ID	Provider	CPU cores	Memory	HDD	Machines	Country
1	TUT	1	2GB	14 GB	4	Finland
2	AIMES	4	4GB	45 GB	1	UK

Table 6.1 *MUSA project hardware setup*

Software-wise, the setup included the resources specified in table 6.2. All machines run on CentOS 7, updates are delivered once per month.

Name	Service type	Auto-managed	ACL	Default location
MoveUs MJP	Web service	y	y	TUT
MoveUs CEC	Web service	y	y	TUT
PostgreSQL	Database	n	n	AIMES
DryWall	Identity manager	y	n	TUT
Oauth2orize	OAuth token issuer	y	n	TUT
TSM Engine	Orchestrator	y	y	TUT

Table 6.2 *MUSA project software setup*

7. CONCLUSIONS AND FUTURE WORK

Even though the framework performed well during the tests, there is still a large room for improvements. Those improvements can be divided into four main groups: deployment process improvements, operation time improvements, user interaction improvement, and system design improvements.

From the deployment point of view, the most time consuming task was the compilation of some components. This installation method also makes it hard to uninstall the component afterwards, since the component itself is not managed by the package manager. It might be feasible to collect the OS versions, download containers with build utilities and package the software on the server beforehand, and then distribute assembled packages between the machines. This will reduce the time needed for the installation, and enable easier component version management. Yet another way is to pre-package the whole framework beforehand and to distribute it as a number of distro-specific packages. This way will allow easier management and update of the framework, but it will require additional work to provide non-stop support, and many distros are getting updates from unstable branches which will be hard to track. Also, it might worth trying to revert to the old SSI-style method, which basically means preparing and bundling the whole system as an image ready to deploy on the target machine. If provided with a both GUI and script-based interfaces, and ability to upload the image to the machine using PXE, it can make a good solution for the data center.

Main operation time improvements that can be proposed are the template engine issue, and the JWT security checkup mechanism. As the real-time configuration manager, both confd and Consul_template work fine, but their main drawback is the use of Go's text/template engine. Go text/template is very simple yet very ugly, and it has very limited functionality in a way it is used. For example, it is impossible to check whether the array contains particular value. The only option provided is to go item by item, performing some task if the match is found, and there

is no solution to mitigate duplicate cases. For this case, the manual recommends to use structures, but as it takes recompilation, it is virtually impossible to prepare structure templates for each and every possible service. For this reason, some small custom poller can be implemented in almost any interpreted language, such as Ruby, Python, Perl, or even Node.JS. The only two functions needed are JSON processing and template handling.

The JWT processing mechanism is also an improvement to be considered. JWT claims can be a powerful source to check whether the particular request should be allowed for further processing or not. Lua language used by Nginx can push those possibilities even further, as it allows the script to query files, directories, databases, remote servers, and many other sources. Thus, this script can be expanded to include more sophisticated checkups. Yet, it might be difficult to configure it correctly for someone who is not familiar with Lua, as the checkup script is configured manually. Since the checkup process can be represented as a linear flow is in most cases, it might be not so hard to create a script generator that will use some UML-based diagram processing system, for example PlantUML. That said, the JWT mechanism might be improved in both performance and user interaction sides.

To make user interaction easier, it might be a good idea to create a unified WebUI that aggregates features from both clustering and directory REST interfaces, to provide a single control point over the whole infrastructure. It should not be very hard as all the systems used provide at least some kind of web interface, but the main problem is the visual design of the UI. Pacemaker's pcsd interface alone had a number of options, edit fields and drop-down lists, and with more possibilities more complexity can come. The UI can be divided into multiple workflows, for example by the service nature, so that it will not be overloaded with additional options too much.

It also might be a good idea to switch to HTTP/2 for HAproxy-Nginx connections by default, as it will allow use of long running connections, decreasing the time overhead required for the TCP three-way handshake between those components. HAproxy-Nginx connections are the only framework-managed connections that go between physical nodes, so the use of long running tunnel can provide additional speedup, removing the largest part of the network overhead. With this improvement it might be even possible that for physically distant servers the framework-managed connection will perform faster than the "bare system" one.

BIBLIOGRAPHY

[1] Inc. Adaptive Computing. *Torque 6.0.2 Administrator Guide*, August 2016.

[2] T. Anderson and J. C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, May 1983.

[3] Amnon Barak and Amnon Shiloh. *MOSIX Administrator's, User's and Programmer's Guides and Manuals*, 2015.

[4] Amnon Barak and Amnon Shiloh. The mosix cluster management system for distributed computing on linux clusters and multi-cluster private clouds. Technical report, 2016.

[5] Chi-Chung Cheung, Man-Ching Yuen, and A. C. H. Yip. Dynamic dns for load balancing. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 962–965, May 2003.

[6] E. Dilger, R. Karrelmeyer, and B. Straube. Fault tolerant mechatronics [automotive applications]. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, pages 214–218, July 2004.

[7] Navendu Jain; Srikanth Kandula; Changhoon Kim; Parantap Lahiri; Dave Maltz; Parveen Patel; Sudipta Sengupta; Albert Greenberg. Vl2: A scalable and flexible data center network. Association for Computing Machinery, Inc., August 2009.

[8] Qusay F. Hassan. Demystifying cloud computing. *CrossTalk*, pages 16–21, January 2011.

[9] Y. S. Hong, J. H. No, and S. Y. Kim. Dns-based load balancing in distributed web-server systems. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, pages 4 pp.–, April 2006.

[10] National Instruments. Understanding raid. `http://www.ni.com/white-paper/7665/en/`, April 2012.

[11] Jen-Hao Kuo, Siong-Ui Te, Pang-Ting Liao, Chun-Ying Huang, Pan-Lung Tsai, Chin-Laung Lei, Sy-Yen Kuo, Yennun Huang, and Zsehong Tsai. An evaluation of the virtual router redundancy protocol extension with load balancing. In *11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 7 pp.–, Dec 2005.

[12] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. Openmosix, openssi and kerrighed: a comparative study. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 1016–1023 Vol. 2, May 2005.

[13] K. C. Markham and R. A. Milliken. Software fault tolerance for a flight control system. In *Computers and Safety, 1989. A First International Conference on the Use of Programmable Electronic Systems in Safety Related Applications*, pages 18–22, Nov 1989.

[14] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. Towards an efficient single system image cluster operating system. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 370–377, Oct 2002.

[15] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). Technical Report UCB/CSD-87-391, EECS Department, University of California, Berkeley, Dec 1987.

[16] L. Perkov; N.Pavkovic; J. Petrovic. High-availability using open source software. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 167–170, May 2011.

[17] G. F. Pfister. The varieties of single system image. In *Advances in Parallel and Distributed Systems, 1993., Proceedings of the IEEE Workshop on*, pages 59–63, Oct 1993.

[18] Tara M. Swaminatha and Charles R. Elden. *Wireless Security and Privacy: Best Practices and Design Techniques.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-6(6):585–594, Nov 1980.

[20] Mark Verber. Security rules of thumb. http://www.verber.com/mark/cs/security/rules-of-thumb.html.

[21] Wikipedia. Computer cluster — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Computer%20cluster&oldid=726174997, 2016. [Online; accessed 24-July-2016].

[22] Qingwei Yang, Zu-Kuan Wei, and J. H. Kim. A hot backup sip server system based on vrrp. In *Networked Computing and Advanced Information Management (NCM), 2010 Sixth International Conference on*, pages 17–20, Aug 2010.

A. SERVICE CONFIGURATION TEMPLATES

This appendix contains service configuration templates for Consul_template. Templates are written in Go Template language.

A.1 HAproxy config template

```
global
    log 127.0.0.1   local0
    log 127.0.0.1   local1 notice
    maxconn         4096
    user            haproxy
    group           haproxy
    daemon
    stats           socket /var/run/haproxy.sock mode 600 level admin

defaults
    log     global
    mode    http
    option  httplog
    option  dontlognull
    option log-health-checks
    retries 3
    option  redispatch
    maxconn 2000
    timeout connect 1000
    timeout client  5000
    timeout server  5000
    option  http-server-close

frontend stats
    bind  *:8888
    stats enable
    stats uri /
    stats auth user:pass
```

```
# HTTP MAPPING
frontend generic_rest
    mode http
    bind *:8185
    option  forwardfor
    {{ range $service := services }}{{/*
    */}}{{ range $tag := .Tags}}{{/*
    */}}{{ if eq $tag "http" }}{{/*
    */}}acl {{$service.Name}}_tag path_beg /{{$service.Name}}{{/* BREAK */}}
    use_backend {{$service.Name}}_backend if {{$service.Name}}_tag
    {{ end }}{{ end }}{{ end }}

{{ range $service := services }}{{/*
*/}}{{ range $tag := .Tags }}{{ if eq $tag "http" }}{{/*
*/}}backend {{$service.Name}}_backend
    mode http
    balance leastconn
    option httpclose
    option forwardfor
    cookie JSESSIONID prefix
    option tcp-check {{/*
    reqrep ^([^\ ]*\ /){{.}}[/]?(.*)     \1\2 */}}
    {{ range service $service.Name }}{{/*
     */}}server {{ .Node }} {{ .Address }}:{{ .Port }} check-ssl {{/*
     */}}ssl verify required crt /etc/haproxy/client.pem {{/*
     */}}ca-file /etc/haproxy/cacert.pem check{{/* LINE BREAK */}}
    {{ end }}{{/*
    */}}
{{ end }}{{ end }}{{ end }}

# TCP MAPPING
{{ range $service := services }}{{/*
*/}}{{ range $tag := .Tags }}{{ if eq $tag "tcp" }}{{/*
*/}}{{ $port := key_or_default (print "service/" $service.Name) "0" }}{{/*
*/}}{{ if gt $port "0" }}{{/*
*/}}listen {{$service.Name}} :{{ $port }}
    mode tcp
    balance roundrobin
    option tcplog
    option tcp-check
    {{ range service $service.Name }}{{/*
     */}}server {{ .Node }} {{ .Address }}:{{ .Port }} {{/*
     */}}check port {{ .Port }}{{/* LINE BREAK */}}
    {{ end }}{{/*
```

```
    */}}
{{ end }}{{ end }}{{ end }}{{ end }}
```

A.2 Nginx config template

```
lua_package_path "/etc/nginx/lua/?.lua;;"; {{/*

*/}}{{ range services }}{{/*
*/}}{{ range $service := service .Name }}{{/*
*/}}{{ range $map_tag := $service.Tags }}{{/*
*/}}{{ if eq $map_tag "map" "automap" }}{{/*
*/}}{{ range $http_tag := $service.Tags}}{{/*
*/}}{{ if eq $http_tag "http" }}{{/*
*/}}{{ if eq $service.Node "<%= Socket.gethostname %>" }}
server {
  listen {{ $service.Address }}:{{ $service.Port }};
  ssl on;
  server_name <%= Socket.gethostname %>;

  ssl_certificate         /etc/nginx/ssl/server.crt;
  ssl_certificate_key     /etc/nginx/ssl/server.key;
  ssl_client_certificate /etc/nginx/ssl/cacert.pem;
  ssl_verify_client       on;

  access_by_lua 'require("jwt-check").check_token("{{ $service.Name }}")';

  location / {
    proxy_pass http://127.0.0.1:{{ $service.Port }};
    proxy_set_header X-ClientCert-DN $ssl_client_s_dn;
    proxy_set_header Host $http_host;
    proxy_set_header X-Forwarded-Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
{{end}}{{end}}{{end}}{{end}}{{end}}{{end}}{{end}}
```

B. NGINX LUA SCRIPT FOR JWT ACL CHECK

```
local _f = {}

function _f.check_token(service)
  -- Fix package path
  package.path = package.path .. ";/usr/local/share/lua/5.1/?.lua" ..
    ";/usr/local/share/lua/5.1/?/init.lua"

  -- Load libs
  local cjson = require "cjson"
  local jwt = require "resty.jwt"
  local validators = require "resty.jwt-validators"
  local resty_consul = require "resty.consul"
  local consul = resty_consul:new({
    host = "127.0.0.1",
    port = 8500
  })

  -- Check if the token is needed
  local token_needed = consul:get_decoded(
    string.format("/kv/claims/%s",service))
  if token_needed then
    -- Open public key file
    local public_key_file = io.open("/etc/nginx/lua/certificate", "r")
    if not public_key_file then
      ngx.log(ngx.ERR, "Can't open public key")
      return ngx.exit(503)
    end

    -- Read public key
    local public_key = public_key_file:read "*a"
    public_key_file:close()

    -- Get token
    local token = ngx.req.get_headers()["Authorization"]
    token = token:gsub("Bearer ", "")
```

```
    if not token then
      ngx.log(ngx.ERR, "Token missing")
      return ngx.exit(403)
    end

    -- Verify token
    jwt:set_alg_whitelist({ RS256 = 1 })
    local jwt_obj = jwt:verify(public_key, token)
    if not jwt_obj["verified"] then
      ngx.log(ngx.ERR, "Bad token")
      ngx.exit(403)
    end

    -- Check if some claims are needed
    local claims_needed = consul:get_decoded(
      string.format("/kv/claims%s", ngx.var.request_uri))
    -- Fill out claims table, each claim is a word
    claims_table = {}
    claims_needed[1].Value:gsub("[A-Za-z_]+",
      function(c) table.insert(claims_table, c) end)
    -- Set up verification
    local jwt_obj = jwt:verify(public_key, token, {
      authorities = function(val)
        -- Check if the claim is in authorities array
        for _,claim_needed in ipairs(claims_table) do
          for _,claim_provided in ipairs(val) do
            if claim_provided == claim_needed then return true end
          end
        end
        -- Return false if the claim was not found
        return false
      end
    })
    -- Check if the token is valid
    if not jwt_obj["verified"] then
      ngx.log(ngx.ERR, "Bad token")
      ngx.exit(403)
    end
    -- End of checks
  end
end
return _f
```

C. LOAD BALANCING TEST COUNTER

This is a simple test script, which counts the amount of processed requests. The script is written in PHP.

```
<?php
$c_file = "counter";
// Create counter file if it does not exist
if (!file_exists($c_file)) {
  $f = fopen($c_file, "w");
  fwrite($f,"0"); // Initial value
  fclose($f);
}

// Read the current value of our counter file
$f = fopen($c_file,"r");
$cVal = fread($f, filesize($c_file));
fclose($f);

// Write the new value into file
$cVal++;
$f = fopen($c_file, "w");
fwrite($f, $cVal);
fclose($f);
echo "$cVal\n";
?>
```